Commando: Solution

Solution 1

$O(n^3)$: Using dynamic programming, let $f(n)$ indicate the maximum battle effectiveness after adjustment. We have transfer equations below:

$$f(n) = \max_{0\le i<n} \left\{ f(i) + g(\sum_{j=i+1}^{n} X_j) \right\}, g(x) = Ax^2 + Bx + C$$

Such a solution should scored around 20 %.

Solution 2

$O(n^2)$: Use pre-calculated partial sum to accelerate the solution above. Let $S_i = \sum_{i=1}^{n} X_i$, we have $f(n) = \max_{0\le i<n} \{f(i) + g(S_n - S_i)\}$

Such a solution should scored around 50 %.

Solution 3

$O(n)$: Consider two decisions $i < j$, we choose i instead of j if and only if :

$$\begin{aligned}
&f(i) + g(S_n - S_i) > f(j) + g(S_n - S_j) \Leftrightarrow \\
&g(S_n - S_i) - g(S_n - S_j) > f(j) - f(i) \Leftrightarrow \\
&A\left((S_n - S_i)^2 - (S_n - S_j)^2\right) + B\left((S_n - S_i) - (S_n - S_j)\right) > f(j) - f(i) \Leftrightarrow \\
&A(2S_n - S_i - S_j)(S_j - S_i) + B(S_j - S_i) > f(j) - f(i) \Leftrightarrow \\
&A(2S_n - S_i - S_j) + B > \frac{f(j) - f(i)}{S_j - S_i} \Leftrightarrow \\
&2S_n < \frac{1}{A}\left(\frac{f(j) - f(i)}{S_j - S_i} - B\right) + S_i + S_j
\end{aligned}$$

Thus, we can use a queue holding all necessary decisions. Each decision in the queue is the best decision during a specific range of S_n. When a decision is to be made, we simply begin searching at one end of the queue, find the first decision where S_n is in its range. After that, we put decision N at the other end of the queue as a new decision, updating range of decisions before it using the inequality above.

Such a solution should scored 100 %.

Patrol: Solution

The road network forms a tree T. A tree with N nodes has $N-1$ edges. In T, the length of a tour that visits all edges is $2(N-1)$, because each edge is visited twice. Recall that adding edges into a tree creates cycles.

Simpler case

We consider a simpler case when $K = 1$. Suppose that we add edge e to T. The resulting graph T' contain exactly one cycle C. The cheapest tour visiting all edges uses each edge in C once and all other edges twice. Denote $C - e$ as path P. The new length of the required tour is

$$2(N-1) - L + 1$$

where L is the length of P. Thus, for $K = 1$, we need to find the maximum path length for paths in T. This value is called the *diameter* of T.

There are many ways to find the diameter. We shall use dynamic programming, which can be turn to be the solution for the general case.

First we root the tree at some node r; the parent-child relation between adjacent nodes can be defined naturally. For each node u, let $H[u]$ denote the length of the longest path from u to some of its descendants. We can compute $H[u]$ for each u, in $O(N)$ time, using a simple dynamic programming.

Consider the longest path P, let node u be the node on P closest to the root r. By definition, u is unique. Given u, the length of P must be either $H[u]$, if u has one children, or

$$\max_{v,w\ :\ \text{different children of } u} (2 + H[v] + H[w])$$

when u has more than one children. The value above is important to the case where $K = 2$ as well, so let's define it as $L[u]$. Formally, $L[u]$ is the maximum length of paths containing u such that u is the closest node to root r.

Thus by enumerating all nodes, one can find the length of the longest path; thus, one can compute the answer to the case where $K = 1$.

When $K = 2$

Let's call both edges e_1 and e_2. Let path P_i be a unique path that join two endpoints of e_i, also let's call a unique cycle induced by adding each edge e_i (separately) as C_i. Note that C_i is a union of P_i and e_i.

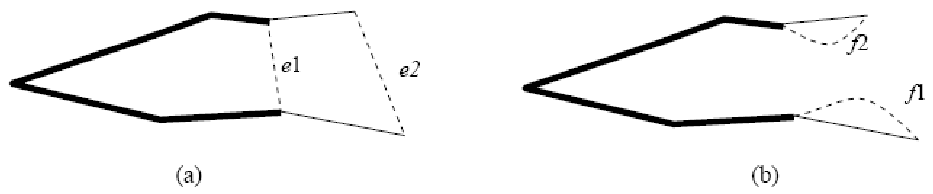


Figure 1: (a) Edges e_1 and e_2 are shown as dashed lines. Paths P_1 and P_2 intersect. The intersection is shown as thick line. In the tour, these edges must still be traversed over twice. (b) The new edges f_1 and f_2 are shown as dashed lines. Note that the number of times each edge on the tree is traversed on is the same as before.

When P_1 and P_2 are disjoint, the length of the desired tour that traverses all edges is

$$2(N-1) - |L_1| - |L_2| + 2$$

where L_i is the length of P_i.

It gets more complicated when P_i's intersect. However, since one must traverse on each e_i exactly once, it is not hard to prove the following claim.

Claim: If P_1 and P_2 intersects, there is another pair of edges f_1 and f_2 such that the paths joining each edge's endpoints are disjoint, and the length of the tour traverses all edges in $T + f_1 + f_2$ is the same as in $T + e_1 + e_2$.

The proof is left out, but Figure 1 illustrates the idea of the proof.

From the claim, to find how to add two edges to minimize the tour, we need to only consider finding a pair of disjoint paths whose sum of lengths is maximum. This, again, can be solved using dynamic programming in $O(N)$ time.

Beside $H[u]$, we need other variables. Let T_u be the subtree rooted at u. We define:

- $A[u]$ is the maximum length of paths inside T_u.
- $B[u]$ is the maximum sum of lengths of any pairs of edge-disjoint paths P and Q in T_u such that one endpoint of P is u.

Figure 2 shows examples of paths considered in $A[u]$ and $B[u]$.

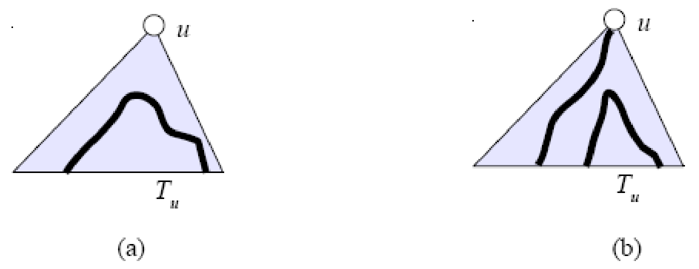


Figure 2: (a) Paths considered in $A[u]$. (b) A pair of paths considered in $B[u]$.

Let $ch(u)$ denote the number of children of u on the rooted tree T. It takes $O(ch(u))$ time to compute $A[u]$ from information from its children by taking the maximum of $A[v]$ for all children v of u and $L[u]$.

To compute $B[u]$, a straight-forward implementation takes $O(ch(u)2)$ time. A careful implementation only takes $O(ch(u))$ time. (See discussion in the next section.)

With A's and B's of all child nodes of u at hand, one can find $D[u]$ the maximum sum of lengths of pairs of paths P_1 and P_2 such that

- P_1 and P_2 are disjoint,
- P_1 contains u, and
- Among all nodes in P_1 and P_2, u is the closest to root r of T.

Again, a careful implementation runs in $O(ch(u))$ time. Easier implementations that run in $O(ch(u)^2)$ time and $O(ch(u)^3)$ time exist. We discuss the implementations later.

After computing all $O[u]$'s, the minimum length of the desired tour is

$$2(N-1) - \max_u D[u] + 2$$

Computing $B[u]$ and $D[u]$

We first discuss how to compute $B[u]$. Let $CH(u)$ denote u's children. Recall that $B[u]$ is the maximum sum of the length of a pair of edge-disjoint paths P and Q such that u is one end of P.

There are many cases to for P and Q:

- Case 1: Both P and Q contains u. In this case, we can compute $B[u]$ by finding 3 children with largest hight.
- Case 2a: P contains edge (u, v) for some child v in $CH(u)$, and Q also lies entirely in T_v. In this case, we have that $B[u] = 1 + B[v]$.
- Case 2b: P contains edge (u, v) for some child v in $CH(u)$, but Q lies entirely in T_w for some child w not equal v. In this case, $B[u] = 1 + H[v] + A[w]$.

Case 1 and Case 2a can be considered in $O(ch(u))$ time. By checking all pairs of children in $CH(u)$, we can consider Case 2b in $O(ch(u)^2)$ time. The time can be reduced to linear by noticing that we can preprocess by finding a child x with maximum $A[x]$. With that, we can consider the value of $1 + H[v] + A[x]$ when v is not equal to x, and $1 + H[x] + \max_{w \neq x} A[w]$ when $v = x$. The total running time is $O(ch(u))$.

The same idea can be applied to computing $D[u]$. In this case we want to find two edge-disjoint paths P and Q in T_u. There are 3 cases to consider:

- Both P and Q contain u.
- Neither P nor Q contain u.
- One contains u.

The first two cases are easy to implement to run in time $O(ch(u))$. The last one can be implemented to run in $O(ch(u)^3)$. The idea from the computation of $B[u]$ can be applied here to reduce the running time to $O(ch(u)^2)$ and $O(ch(u))$.

Scoring

Since optimizing the computation of B's and D's are not the essential part of the task, solutions that uses both $O(ch(u)^3)$ and $O(ch(u)^2)$ per node u should score the majority of the test cases.

Signaling: Solution

Solution

The problem description is: Given N points on 2D plane, we guarantee that any three points are not in a line and any four points are not on a circle. Randomly pick 3 different points to make a circle, compute the average number of points that are inside or on the circle. We assume that each point has the same probability to be chosen.

Our goal is to compute the average number of points that are covered by the circle. In fact, we only need to compute the sum of points that are covered by any circle and divide the result by $\binom{N}{3}$ to get the exact answer for our problem.

Algorithm 1: A brute force solution for this problem is to enumerate any three different points and calculate their corresponding circumcircle. Then we can count how many points are inside or on the circle in $O(N)$ time. Clearly, this solution works in time $O(N^4)$.

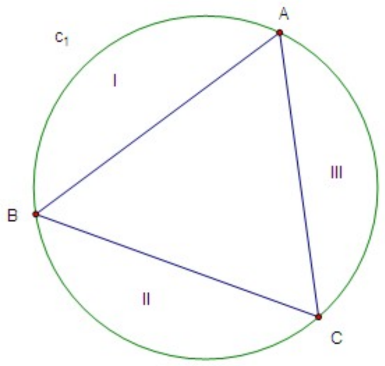
Algorithm 1 $O(N^4)$

Require: a set of N points $P_i = (x_i, y_i)$.
Ensure: return the average number of points which are covered by the circle.

```
1:  $ans \leftarrow 0$
2:  for $i = 1$ to $N$ do
3:      for $j = i + 1$ to $N$ do
4:          for $k = j + 1$ to $N$ do
5:              $(x, y, r) \leftarrow \textsc{ComputeCircumcircle}(P_i, P_j, P_k)$
6:              for $l = 1$ to $N$ do
7:                  if the distance between $A_l$ and $(x, y)$ is at most $r$ then
8:                      $ans \leftarrow ans + 1$
9:                  end if
10:             end for
11:         end for
12:     end for
13: end for
14: return $ans/\binom{n}{3}$
```

Now our task is to count the number of quadruplets $(A, B, C : D)$ such that point D is inside the circumcircle of A, B, C. Let's consider the position relation between D and A, B, C:

Seeing the below figure, we can deduce that if point D is in area I, II or III, then the sum of the angle D and the angle of the opposite point must be $>$ 180 degree since interior angle is always larger than the corresponding circumferential angle. For example, if D is in area I, then $\angle ADB + \angle ACB > 180°$ and the four points must form a convex quadrangle. Similarly, if D is outside the circle, the sum of the angle D and the corresponding opposite angle must be $< 180°$. And another case is D is inside the triangle ABC, then the four points must form a concave quadrangle.



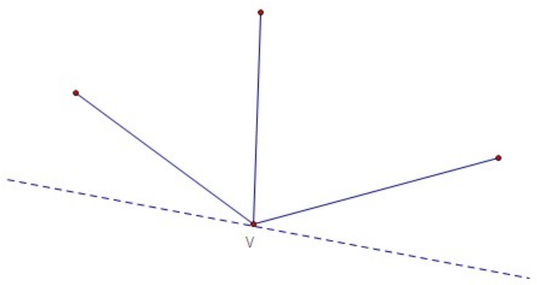
Hence let's consider every quadrangle we get from the set of points, note that two quadrangles are distinct if and only if the set of the 4 points are different(4 points can construct different quadrangles if one point is inside the triangle of another 3 points). We can see that

1. if it's convex, then there are 2 ways to pick 3 points to make a circle so that the circle contains the remaining point. This is because the sum of the 4 angles is equal to 360 degree and no 4 points are on a circle, there must be exactly a pair of opposite angles such that their sum is greater than $180°$. For example, if $\angle A + \angle C > 180°$ in a quadrangle $ABCD$, then point C must be in the circle of ABD while A must be in the circle of BCD.

2. if it's concave, then there is exactly 1 way to pick 3 points to make a circle such that the circle contains the remaining point. In this case, the 4th point must be in the triangle of the other 3 points.

Let P = the number of convex quadrangles and Q = the number of concave quadrangles. Our new goal is to compute the value of $2P + Q$ and it is straightward that $P + Q = \binom{N}{4}$. There are many ways to compute P or Q or any linear combinations of P and Q in $O(N^2 \log N)$ or $O(N^3)$ time, then we can get $2P+Q$. Next we will give a very simple method to compute $4P + 3Q$.

For any quadrangle T and a vertex v of T, if we can draw a line through v such that all other 3 points lie in one side of this line, then we say that the vertex v is "good".



We can see that for any convex quadrangles, all vertices are "good" while there are only 3 "good" vertices in a concave quadrangle except the innermost point. If we can count the number of "good" points for any 4 points, then we have got the value of $4P + 3Q$.

Algorithm 2: enumerate a point as v, and sort all other points by their polar angles with respect to v. Then we can scan all other points in the order and maintain the number of points such that the angle between it and the current point is $< 180°$. Then we can count the number of quadrangles in which v is a "good" vertex in $O(N)$ time. For each point v, it takes $O(N \log_2 N)$ time to sort all other points by polar angles and $O(N)$ time to scan all points to get the answer. So the total running time is $O(N^2 \log_2 N)$. Then we have got algorithm 2:

Algorithm 2 $O(N^2 \log_2 N)$

Require: a set of N points $P_i = (x_i, y_i)$.

Ensure: return the average number of points which are covered by the circle.

```
1: $ans \leftarrow 0$
2: for $i = 1$ to $N$ do
3:     Sort all other points by the polar angle respect to point $P_i$ and get $P'_1, P'_2, \ldots, P'_{N-1}$    ▷ $O(N \log_2 N)$
4:     $k \leftarrow 1, s \leftarrow 0$
5:     for $j = 1$ to $N - 1$ do
6:         while $\vec{P_iP'_k} \times \vec{P_iP'_j} \leq 0$ and $s < N - 1$ do
7:             $s \leftarrow s + 1$
8:             $k \leftarrow next(k)$    ▷ $next(N-1) = 1$
9:         end while
10:        $ans \leftarrow ans + \frac{(s-1)(s-2)}{2}$
11:    end for
12:    $s \leftarrow s - 1$
13: end for
14: $ans \leftarrow ans - 2\binom{n}{4}$    ▷ $(4P + 3Q) - 2(P + Q) = 2P + Q$
15: return $ans/\binom{n}{3}$
```

Till now, we have got a complete algorithm in $O(N^2 \log_2 N)$ time. In fact, there are many similar methods to compute a linear combination of P and Q and after we sort all other $N-1$ points, we can enumerate two points to calculate the result to get an $O(N^3)$ algorithm which can pass 70% of the test cases.