USACO DEC08 Problem 'fence' Analysis

by Bruce Merry and Jacob Steinhardt

We start by noting a few naive solutions that run in either N^5 or N^4 time.

Throughout this discussion, we will let the "hull" denote a convex set of points that we are trying eventually to complete to form a convex polygon. All efficient algorithms work by starting with a hull of one or two points and looking at ways to enlarge that hull up to a convex polygon.

We can do an N^5 DP by letting the state be (p_1,p_2,p_3,p_4), where p_1 and p_2 are the first two points in the hull, and p_3 and p_4 are the last two points we added to the hull. We update the DP by seeing what possible next points we can use. By looping through all O(N) possible next points, we get an O(N^5) DP. This works because the first and last two points in the hull are sufficient to check that the next point doesn't break the convexity condition.

We can speed this up to N^4 by requiring that our first point be the bottom-most point in the hull. Then we only need to keep track of the first point and the last two points in order to check for convexity. This yields an $O(N^4)$ solution. However, we can do better.

The following is an $O(N^3)$ solution. Start by fixing the bottom-most point B (bottom-left if there is a tie). The subsequent points must all follow in sequence in the angle they make with B. Thus, if we start by sorting all the other points by the angle they make with B, there is a relatively simple recursive solution: pick the block counter-clockwise of B, then continue round 180 degrees, deciding whether to pick each point or not, and also checking that the shape stays convex.

To make this run in a practical amount of time, one can note that during the recursion, only the last two points actually used determine whether future points can be used (for a fixed choice of B). Thus, one can use dynamic programming over this space to reduce the running time to $O(N^3)$.

Depending on how you've implemented it, you may still find this rather tight for the time limit. The situation can be improved by precomputing some information about which range of values is potentially valid given the previous two vertices, independently of B. That moves the expensive geometric computation out of the inner-most loop.

More formally, we can recurse (using memoization) with the state being (p_1,p_2,p_3), where p_1 is the point we started at, p_2 is the point we last used in our polygon, and p_3 is the next point we are considering. Then we have two choices. We can choose to take the point p_3 as our next point in the polygon, at which point we go to $(p_1,p_3,f(p_2,p_3))$, where $f(p_2,p_3)$ is the point closest to the line p_2p_3 (when measured with respect to counterclockwise angle about p_3). Otherwise, we choose not to use p_3 as the next point in our polygon, and instead move on to consider the next point about p_2, that is, we go to $(p_1,p_2,g(p_2,p_3))$, where $g(p_2,p_3)$ is the point closest to the line p_2p_3 when measured with respect to counterclockwise angle about p_2. For each of these choices, we need to

first make sure it is actually a valid choice (for example, that such a choice exists and that it doesn't mess up the convexity requirements).

This state space might seem a bit odd, but basically what we are doing is just considering all possible points that we can use for the next point in our polygon in counter-clockwise order. By having a state space consisting of the last point next *potential* point instead of, say, just keeping track of the last two points, we manage to shave a factor of N off of our runtime because we can do O(1) updates instead of O(N) updates (we only need to consider two possible choices instead of going through all possible next points). Intuitively, we are "relaxing" the definition of our recursive function a bit so that it is easier to compute, while still keeping enough structure that we actually get the information we want at the end.

Note that the functions $f(p_2,p_3)$ and $g(p_2,p_3)$ only take two parameters, so we can pre-compute each of their values in linear time and then look them up while actual performing our recursion.

Finally, a note on how points were assigned:

Exponential search (consider all subsets of points and take their convex hull) was meant to receive 5/20.

N^5 received 8/20.

N^4 received about 11/20.

Optimized N^4 was meant to receive 14/20, though some N^4 solutions with heuristics received more.

N^3 receives full points.

Only a few competitors coded up a speedy N^3 solution. All other high-scoring solutions were N^4 solutions that only looped through some fraction of possible bottom points.

Here is sample code:

```
#include <stdio.h>
#include <stdlib.h>
#include <cmath>
#include <complex>
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

typedef long long int LL;
typedef complex<long long int> CL;
typedef long double LD;

#define MAX (305)
```

```
LD one = (LD)1.0;

int N;
CL arr[MAX];           //coordinates
int ind[MAX][2*MAX];   //sorted list of other coordinates around this point
int lkp[MAX][MAX];     //where in a list a given point occurs
int tar[MAX];          //dummy array for sorting

CL srt;

int dp[MAX][2*MAX];
bool mem[MAX][2*MAX];

int next[MAX][MAX];
/* pre-computed list of points
 * next[i][j] is the point that makes the smallest counter-clockwise angle
 * with line IJ when measured about the point j
 */

bool ok(int i,int j,int k){ //returns true if k is strictly to the left of
line IJ
        return imag(conj(arr[j]-arr[i])*(arr[k]-arr[i]))>0;
}

bool ok2(int i,int j){ //returns true if i has height at least that of j
        return imag(arr[j]) >= imag(arr[i]);
}

int nxt(int p3,int p2){
  //finds the next point after p2 (in counterclockwise order about p3)
        int ans = 0;
        while(ans<N-1&&ok(p2,p3,ind[p3][ans])) ans++;
        while(ans<2*N-2&&!ok(p2,p3,ind[p3][ans])) ans++;
        if(ans==2*N-2) return -1;
        else return ind[p3][ans];
}

bool opt=0;

int rec(int p1,int p2,int p3){
        if(p3==p1) return 2; //stop if we get back to our starting point
        if(mem[p2][p3]) return dp[p2][p3];
        dp[p2][p3]=max(
                        ok2(p1,p3)?1+rec(p1,p3,next[p3][p2]):0,
                        //use this point if it's legal
(i.e. above p1)
                        lkp[p2][p3]<2*N-3?rec(p1,p2,ind[p2][lkp[p2][p3]+1]):0
        //go on to next point
if there is one
                );
        mem[p2][p3]=1;
        return dp[p2][p3];
}

inline LD f(CL z){
        return atan2(one*(imag(z)-imag(srt)),one*(real(z)-real(srt)));
```

```
  //compares based on the angle that z makes with respect the point we are
sorting around
}

bool cmp(int i,int j){
        return f(arr[i]) < f(arr[j]);
}

int main(){
        FILE *fin = fopen("fence.in","r");
        FILE *fout = fopen("fence.out","w");
        fscanf(fin,"%d",&N);
        LL x,y;
        for(int i=0;i<N;i++){
                fscanf(fin,"%lld%lld",&x,&y);
                arr[i]=CL(x,y);
        }

        for(int i=0;i<N;i++){
                srt = CL(real(arr[i]),imag(arr[i]));
                for(int j=0;j<i;j++) tar[j]=j;
                for(int j=i+1;j<N;j++) tar[j-1]=j;
                //sort the points based on angle about point i
                sort(tar,tar+(N-1),cmp);
                //fill in indexing and look-up tables
                for(int j=0;j<N-1;j++){
                        ind[i][j]=tar[j];
                        lkp[i][tar[j]]=j;
                }
                for(int j=N-1;j<2*N-2;j++)
                        ind[i][j]=ind[i][j-N+1];
        }

        //pre-compute some more values we'll need
        for(int i=0;i<N;i++)for(int j=0;j<N;j++)
                if(i!=j) next[i][j]=nxt(i,j);

        int best = 0;
        for(int i=0;i<N;i++){ //loop through all possible first two points
                for(int a=0;a<N;a++)
      for(int b=0;b<2*N;b++)
        mem[a][b]=0; //initialize memoization array

    for(int j=0;j<N;j++){
                        if(i==j) continue;
                        if(ok2(i,j)&&next[j][i]!=-1)
         /* check to see if we can actually build a convex polygon
          * starting with i and j that has I as the bottom point
          */
                                best = max(best,rec(i,j,next[j][i]));
                }
        }

        fprintf(fout,"%d\n",best);
        fclose(fin); fclose(fout);
}
```