USACO OPEN07 Problem 'horizon' Analysis

by Richard Ho

Suppose you take the x-coordinates of where each building starts (A_i) and ends (B_i) and sort these 2N numbers. This divides the horizon into (at most) 2N-1 pieces, call them x_1, x_2, x_3, ..., x_k. We know then that the maximum height of the aggregate silhouette is constant in each of the intervals [x_1,x_2], [x_2,x_3], ..., [x_(k-1),x_k]. In fact, if we knew the maximum heights h_1, h_2, ... h_(k-1) for each of the intervals, then the total area is just (x_2-x_1)*h_1 + (x_3-x_2)*h_1 + ... + (x_k-x_(k-1))*h_(k-1). So let us try to find these maximum heights instead.

One way to do this is, for each interval, we go through all the buildings and see which ones contain that interval, and take the tallest of those. However, this is very slow, since for each of the 2N-1 intervals, we would have to go through N buildings, giving us something with a runtime of O(N^2). With N up to 40,000, this is going to be too slow.

Instead we can sweep from left to right and add a height to a list (allowing repetitions) when a building starts, and remove that height from the list when a building ends. The highest number in the list represents the height of the silhouette at that point. So we can just sort the building's start and end locations (making sure to store whether its a start or end, and the associated height). We sweep from left to right, and process all the heights at the same coordinate (buildings can start or end where another building starts or ends) and remember what the final heights were for each coordinate. We know that the height at position x_i is the height of the silhouette on the interval [x_i, x_(i+1)]. Now, if we just used a simple list, each height we process can take up to O(N). But if we used a max-heap (i.e. binary tree where parents are always larger than both children), then each height we process can take up to O(log N), since heap insertions and removals are logarithmic. This would give us a final runtime of O(N log N), which is sufficient.

However, if you really do not like coding data structures from scratch (due to bugs or just plain boredom), then it is worth the effort to learn to use your languages libraries for data structures. Below is a (short) solution from Christos Tzamos that uses STL's map and multiset, which are already sorted containers!

```
#include<cstdio>
#include<vector>
#include<map>
#include<set>
#include<utility>
using namespace std;

map<int,vector<pair<int,int> > > M;
map<int,vector<pair<int,int> > >::iterator it;
vector<pair<int,int> >::iterator i;
multiset<int, greater<int> > S;

int main() {
        int N,L,a,b,h,opens,k;
        long long s=0;
        freopen("horizon.in","r",stdin);
```

```
	freopen("horizon.out","w",stdout);
	scanf("%d",&N);
	while(N--) {
		scanf("%d %d %d",&a,&b,&h);
		M[a].push_back(make_pair(h,1));
		M[b].push_back(make_pair(h,0));
	}
	a = 0;
	for(it=M.begin();it!=M.end();it++) {
		b = it->first;
		if(!S.empty()) s += (long long)(b - a)*(*S.begin());
		a = b;
		for(i=(it->second).begin();i!=(it->second).end();i++) {
			k= i->first;
			opens = i->second;
			if(opens) S.insert(k);
			else S.erase(S.find(k));
		}
	}
	printf("%lld\n",s);
	return 0;
}
```