USACO OPEN09 Problem 'tower' Analysis

by Richard Peng

A O(n^3) DP is fairly straight forward since all that matters is the length of the previous 'row' of hay blocks, which can be expressed as an interval [i..j]. The DP then maximizes the height of the tower given that [i..j] is the bottommost level. Enumerating over the starting point of the next interval [k..i-1] gives the transition.

This can be optimized to only considering two transitions: [i..j]->[i-1..j] (extend the bottom interval for free) and [i..j]->[k*..i-1] where k* is the largest value such that the bottom interval is at least as wide as the one above it. Updating in the proper order (fix i, increase j and decrease k* as we go) gives O(n^2).

To get a subquadratic solution, we need to 'rephrase' the problem. Instead of maximizing height, we use DP to minimize the width of the tower we can build using w_i...w_n instead. To show that this works, we need to make some observations about some values:

Let maxh_i be the maximum height possible using w_i...w_n. Then maxh_i is monotone because we can always take the extra hay and put them on the first row.

Let narrowest_i be the minimum width achievable from i. Then the transition would be:

narrowest_i=sum(w_k:i<=k=narrowest_j)}

We now show that the tower that minimizes the width of the bottom level also maximizes the height. Suppose in the optimal solution for a tower using w_i...w_n, the last level ends at j'. Then j<=j' since j' satisfies sum(w_k:i<=k=narrowest_j'). But we know that maxh_j>=maxh_j'. So taking that transition gives a tower that's at least as tall. So such j gives an optimal solution as well. To make this rigorous, the rest follows by induction.

The above transition for narrowest_i immediately gives a O(n^2). To get a O(nlogn), notice that for each j, all is below a certain value can use it for transition. So we can binary search for such i and update a global minimum for all the available transitions as we enter regions where transitions can be used. This is the intended full-score solution.

This can also be optimized to work in O(n) time, although it's not necessary to get full points. For each j, let the maximum i where the i->j transition can be used be activate_j. Then notice that if we have j1activate_j2, then there is no point of using j2 every (since j1 is always an option when j2 is). So we are left with a monotone queue and we always 'activate' elements of the queue from the right end. This gives O(1) amortized per operation for a total of O(n).

My code for the O(n) solution:

```
#include <cstdio>
#include <cstring>
```

```
using namespace std;

#define FR(i, a, b) for(int i=(a); i<(b); i++)
#define FOR(i, n) FR(i, 0, n)
#define RF(i, a, b) for(int i=(b)-1; i>=(a); i--)
#define ROF(i, n) RF(i, 0, n)

#define MAXN 100011

int n,ans,w[MAXN],s[MAXN];
int q[MAXN],head,tail;
int bes[MAXN],fro[MAXN],barrier[MAXN];

int main(){
  int p,j,p1,del;
freopen("tower.in","r",stdin);
freopen("tower.out","w",stdout);
  scanf("%d",&n);
  s[0]=0;
  FOR(i,n){
    scanf("%d",&w[i]);
    s[i+1]=s[i]+w[i];
  }
  j=n;
  head=0;
  tail=-1;
  ROF(i,n){
    while((head<=tail)&&(s[i]<=barrier[q[head]]))
      j=q[head++];
    fro[i]=j;
    barrier[i]=s[i]*2-s[j];
    while((tail>=head)&&(barrier[q[tail]]<=barrier[i]))
      tail--;
    q[++tail]=i;
  }
  ans=p=0;
  while(p!=n){
    ans++;
    p=fro[p];
  }
  printf("%d\n",ans);
  return 0;
}
```