USACO OPEN10 Problem 'hop' Analysis

by John Pardon

First, we make a number of reductions. Let us look at the sequence of squares that are jumped on on Bessie's return journey. By the rules of the game, no two can be more than K squares apart. Now, knowing this sequence of return squares, Bessie *must* step on each of the squares just past these when she is making her outgoing steps. Now what other squares will Bessie step on during her outgoing journey, supposing we know the sequence of return squares?

Since Bessie is maximizing her total money earned, she obviously will just step on all of the squares with positive monetary value that we haven't already specified her having stepped on. Thus, when we do the computation, it suffices to just keep track of the sequence of squares stepped on during Bessie's return journey, since the sum of the others can be computed quickly from these.

Mathematically, we can express this relation as:

TotalMoneyEarned = (sum over return squares r)(value[r] + value[r + 1]) + (sum of all the intermediate positive values)

Now we can cleverly rewrite this as:

(sum over return squares r)(-abs(value[r])-abs(value[r+1])) + (sum of *all* the positive values up to where Bessie turned around)

(where abs(.) denotes absolute value)

If we fix the place where Bessie turns around, then the second term is determined completely! Thus we can focus on maximizing the first term.

Consider a dynamic programming approach to this problem. We will iteratively calculate the following quantity:

dp[n] = maximum possible value for (sum over return squares r)(-abs(value[r])-abs(value[r+1])) assuming we turn around at square n.

Now it is easy to derive some formula along the lines of:

dp[n] = maximum over all m in the range (n-k,n-1) dp[m] + value[n] + value[n + 1]

Thus with a naive algorithm, we could calculate dp[n] in O(NK) time. However, this is not fast enough. In fact, we can get an O(N log N) time algorithm as follows. The point is that we want to find the maximum over a range (n-k, n) as quickly as possible. This can be done very simply using a heap. Whenever we calculate a new value dp[i], we put it in a heap, which sorts things by dp[i], keeping track of i as well. When we want to find the maximum over all m in the range (n-k,n) of dp[m], we query the heap for the largest value (which takes O(log N) time!). This will give us a large value of dp[i], but i may not be in the range (n-k,n-1). If this occurs, then we just remove this element from the heap and try again. This may happen many times when we are calculating a given value dp[n], so calculating dp[n] could take O(N) time! However, it can't happen too many times, since a particular ordered pair (dp[i],i) only ever enters and exits the heap once. Thus the running time for this algorithm is O(N log N).

There are two additional things one has to be careful of when programming this:

- You can't step on two consecutive squares on your return journey, so the maximum is really over the range (n-k,n-2), so you can't put dp[n] in the heap as soon as you calculate it; you have to delay its insertion a little bit.
- You have to be careful about considering the two possible ways that Bessie could turn around: she could either step to the immediate left of her farthest right square, or she could step farther to the left.

A solution follows:

```
#include <cstdio>
#include <queue>

using namespace std;

int N;
int K;
long long V[300000];
long long part[300000];

struct PQD {
    PQD(){}
    PQD(long long a, int b):value(a),place(b){}
    long long value;
    int place;
    bool operator <(const PQD &o) const {
        return value < o.value;
    }
};

int main(void) {
    FILE *inFile = fopen("hop.in", "rt");
    fscanf(inFile, "%i%i", &N, &K);
    for(int i = 0 ; i < N ; i++) {
        fscanf(inFile, "%lld", &V[i]);

        part[i] = 0;
        if(V[i] > 0) {
            part[i] = V[i];
            V[i] = 0;
        }
        if(i > 0)
            part[i] += part[i - 1];
    }
    fclose(inFile);
```

```
long long best = 0;

priority_queue<PQD> q;
PQD toAdd(0, 0);
bool hasNew = true;
for(int i = 0 ; i < N ; i++) {
    PQD cur;
    PQD ifStepHere;
    bool pushed = false;

    chooseFirst:
    if(q.empty())
        goto chooseSecond;
    cur = q.top();
    if(cur.place + K < i) {
        q.pop();
        goto chooseFirst;
    }

    ifStepHere.place = i;
    ifStepHere.value = cur.value + V[i] + V[i - 1];
    long long pot;
    pot = ifStepHere.value + part[i];
    if(best < pot)
        best = pot;

    pushed = true;
    if(hasNew)
        q.push(toAdd);
    toAdd = ifStepHere;
    hasNew = true;

    chooseSecond:
    if((!pushed) && hasNew) {
        q.push(toAdd);
        hasNew = false;
    }

    if(q.empty())
        goto chooseThird;
    cur = q.top();
    if(cur.place + K <= i) {
        q.pop();
        goto chooseSecond;
    }

    long long pot2;
    pot2 = cur.value + V[i] + part[i];
    if(best < pot2)
        best = pot2;

    chooseThird:

    ;
}
```

```
    FILE *outFile = fopen("hop.out", "wt");
    fprintf(outFile, "%lld\n", best);
    fclose(outFile);

    return 0;
}
```

Taiwan's Willy Liu points out that:

We can even reduce the complexity to O(N) by using a deque to keep the monotonicity. Because the operation on deque is of constant time rather than logarithm one, the complexity will be lower.

Here is my code:

```
#include <cstdio>
#include <algorithm>
#include <utility>

using namespace std;

static const int MAX_N = 250000;
int val[MAX_N];         // score
long long psum[MAX_N];  // partial sum of positive numbers

int main() {
    FILE *fin = fopen("hop.in", "r");
    FILE *fout = fopen("hop.out", "w");
    static pair<int, long long> deque[MAX_N + 1];
         /* the first element of the pair is the index, and the
            second is the max score we can get */

    pair<int, long long> *ds = deque, *de = deque;
    *de++ = make_pair(-1, 0);   // we mark the first element (empty) indexed
-1

    int n;
    fscanf (fin, "%d", &n);
    int k;
    fscanf (fin, "%d", &k);
    long long maxval = 0;                           // the answer to be

    for(int i = 0; i != n; ++i)
        fscanf(fin, "%d", val + i);

    psum[0] = max(0, val[0]);

    for (int i = 1; i != n; ++i)
        psum[i] += psum[i - 1] + max(0, val[i]);

          // calculate the partial sum of positive numbers:
        long long toadd = 0;    /* Record the elements to be pushed
                                   into the deque in the next loop */
    for (int i = 0; i != n; ++i) {
        for (; ds != de && ds->first + k < i; ++ds);  /* pop back
                                if the index > current index - k */
        long long cur = ds->second + val[i] + (i == 0 ? 0 :
                psum[i - 1]) - (ds->first == -1 ? 0 : psum[ds->first + 1]);
        maxval = max(maxval, cur);
        if(i != n - 1) maxval = max(maxval, cur + val[i + 1]);
        if(i != 0) {
            while (ds != de &&
                    toadd > de[-1].second + psum[i] - psum[de[-1].first + 1])
                        // Pop front if no longer be better
                --de;
                *de++ = make_pair(i - 1, toadd);
                        // instead of dp, we push it into the deque
        }
        toadd = cur + val[i + 1];
    }
    fprintf (fout, "%lld\n", maxval);
    return 0;
}
```