Code Jam 2010 - Round 1A

Analysis: Make it Smooth

The Basic Solution

Just about any solution to this problem is going to ultimately rely on building up a smoothed array by first solving smaller sub-problems. The challenge is two-fold: (1) What are the sub-problems you want to solve? And (2) How can you efficiently process them all?

It is natural to start off by making the first `N-1` pixels smooth, and then figuring out afterwards what to do with the last pixel. The catch is what we do with the last pixel `q` depends on `p`, the final pixel we set before-hand. If `p` and `q` differ by at most `M`, then we are already done! Otherwise, we have two choices:

- Delete `q` with cost `D`, leaving the final pixel of our smoothed picture as `p`.
- Move `q` to some value `q'` with cost `|q - q'|`. If `|q' - p| > M`, we will then have to add some pixels before-hand to make the transition smooth. In fact, we will need to add exactly `(|q' - p| - 1)/ M` of these pixels.

Fortunately, both of these cases are easy to analyze, as long as we are willing to loop through every possible value of `q'`.

Perhaps the trickiest part of this setup is understanding insertions. After all, when deciding what steps to take to smooth out the transition from one pixel in the starting image to the next pixel, there are a lot of options: we could change either pixel and we could have any number of insertions between them. The insight is that once we have decided where to move both pixels, it is obvious how many insertions we need to do.

The pseudo-code shown below recursively finds the minimal cost to make `pixels[]` smooth, subject to the constraint that the final pixel in the smoothed version must equal `final_value`:

```
int Solve(pixels[], final_value) {
  if (pixels is empty) return 0

  // Try deleting
  best = Solve(pixels[1 to N-1], final_value) + D

  // Try all values for the previous pixel value
  for (all prev_value) {
    prev_cost = Solve(pixels[1 to N-1], prev_value)
    move_cost = |final_value - pixels[N]|
    num_inserts = (|final_value - prev_value| - 1) / M
    insert_cost = num_inserts * I
    best = min(best, prev_cost + move_cost + insert_cost)
  }
  return best
}
```

To answer the original problem, we just take the minimum value from `Solve` over all possible choices of `final_value`.

Unfortunately, this algorithm will be too slow if implemented exactly like this. Within each call to `Solve`, we are making 257 recursive calls, and we might have to go 100 levels deep. That won't finish in any of our life times, let alone within the time limit! Fortunately, the only reason it is this slow is because we are duplicating work. There are only `256 * N` different sets of parameters that we will ever see for the `Solve` function, so as long as we store the result in a cache, and re-use it when we see the same set of parameters, everything will be much faster. This trick is called Dynamic Programming or more specifically, Memoization.

The Fancy Solution

The run-time of the previous solution is `O(256 * 256 * N)`, which is plenty fast in a competitive language. (Some interpreted languages are orders of magnitude slower at this kind of work than compiled languages - beware!) The extra 256 factor comes from the fact that we need to try 256 possibilities within each function call. It is actually possible to solve this problem in just `O(256 * N)` time. Here are some hints in case you are interested:

- As before, you want to calculate `Cost[n][p]`, the cost of making the first `n` pixels smooth while setting the final pixel to value `p`. Unlike before, you want to do a batch of these updates at the same time. In particular, you want to simultaneously calculate *all* values for `n+1` given the values for `n`.
- So how do we do this batch update? First, let's do an intermediate step to calculate `Cost'[n][p]`, the minimum cost for each value after doing all insertions between pixel `n` and pixel `n+1`. To make this more tractable, it helps to notice that there is never any need to insert a pixel with distance less than `M` from the previous pixel. (Do you see why?)
- The real challenge is that when calculating `Cost[n][]` from `Cost'[n][]`, you are going to want to take minimums over several elements. For example, `Cost[n][q] = min(Cost[n-1][q], {Cost'[n][q-M], Cost'[n][q-M+1], Cost'[n][q-M+2], ..., Cost'[n][q+M]})`. In other words, given the array `Cost'[n][]`, you are going to need to be able to calculate in linear time the minimum element in each range of length `2M+1`. This is an interesting and challenging problem in its own right, and we encourage you to think it through! For a more thorough discussion of this sub-problem, see here.