Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Sivan Toledo and Alan Edelman

Wednesday, April 22nd
Handout 14

Problem Set 6

This problem set is divided into two parts: Part A problems are programming tasks, and Part B problems are theory questions.

Part A questions are due **Tuesday, May 5th** at **11:59PM**.

Part B questions are due **Thursday, May 7th** at **11:59PM**.

Solutions should be turned in through the course website in PDF form using LaTeX or scanned handwritten solutions.

A template for writing up solutions in LaTeX is available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A: Due Tuesday, May 5th

1. **(25 points)** Image Resizing

   In a recent paper, "Seam Carving for Content-Aware Image Resizing", Shai Avidan and Ariel Shamir describe a novel method of resizing images. You are welcome to read the paper, but we recommend starting with the YouTube video:

   `http://www.youtube.com/watch?v=vIFCV2spKtg`

   Both are linked to from the Problem Sets page on the class website. After you've watched the video, the terminology in the rest of this problem will make sense.

   If you were paying attention around time 1:50 of the video, then you can probably guess what you're going to have to do. You are given an image, and your task is to calculate the best vertical seam to remove. A *vertical seam* is a connected path of pixels, one pixel in each row. We call two pixels *connected* if they are vertically or diagonally adjacent. The *best* vertical seam is the one that minimizes the total "energy" of pixels in the seam.

   For some reason, the video didn't spend much time on the most interesting part—dynamic programming—so here's the algorithm:

   Subproblems: For each pixel (i, j), what is the lower-energy seam that starts at the top row of the image, but ends at (i, j)?

   Relation: Let `dp[i,j]` be the solution to subproblem (i, j). Then

   ```
   dp[i,j] = min(dp[i,j-1],dp[i-1,j-1],dp[i+1,j-1]) + energy(i,j)
   ```

Analysis: Solving each subproblem takes $O(1)$ time: there are three smaller subproblems to look up, and one call to `energy()`, which all take $O(1)$ time. There is one subproblem for each pixel, so the running time is $\Theta(A)$, where A is the number of pixels, i.e., the area of the image.

Find `ps6_image.py`. You will need to installed PIL (Python Imaging Library, freely available from `http://www.pythonware.com/products/pil/`), and Tkinter if you want to view images. If you are using Athena (Linux), `add -f 6.006` and run `python2.5`.

In `ResizeableImage.py`, write a function `best_seam(self)` that returns a list of coordinates corresponding to the cheapest vertical seam to remove, e.g., $[(5,0),(5,1),(4,2),(5,3),(6,4)]$. You should implement the dynamic program described above in a bottom-up manner.

`ResizeableImage` inherits from `ImageMatrix`. You should use the following components of `ImageMatrix` in your dynamic program:

- `self.energy(i,j)` returns the energy of a pixel. This takes $O(1)$ time, but the constant factor is sizeable.
- `self.width` and `self.height` are the width and height of the image, respectively.

Test your code using `test_image.py`, and submit `ResizeableImage.py` to the class website. You can also view your code in action by running `gui.py`. Included with the problem set are two differently sized versions of the same sunset image. If you remove enough seams from the sunset image, it should center the sun.

Also, please try out your own pictures (most file formats should work), and send us any interesting before/after shots.

2. **(25 points)** Making Progress

   You work on your thesis over the weekend, and every time you make a change to your code, you run your `test_awesomeness.py` script, which spits out a score telling you how awesome your code is. During two hard days of work, you accumulate a large, time-ordered list of these awesomeness scores, e.g., $[32, 31, 46, 36, 32, 36, 30, 33, 22, 38, 2, 13]$.

   You have a weekly meeting with your advisor, and each week you have to show that you made progress, so that he'll leave you alone for another week. You devise a plan in which every week you will show your advisor a newer version of your code, along with an awesomeness score that is better than the previous week's. To maximize the number of weeks of slacking you get out of your two days of work, you need to calculate a longest increasing subsequence of your awesomeness scores. In the example, one such subsequence would be $[31, 32, 36, 38]$. The subsequence should be strictly increasing, because you need to show improvement each time.

   (a) **(4 points)** Clearly state the set of subproblems that you will use to solve this problem.

(b) **(4 points)** Write a recurrence relating the solution of a general subproblem to solutions of smaller subproblems.

(c) **(2 points)** Analyze the running time of your algorithm, including the number of subproblems and the time spent per subproblem.

(d) **(15 points)** Write a function `longest_increasing_subsequence(scores)` in `progress.py`, which takes a list of scores, and returns the longest (strictly) increasing subsequence of those scores. Write it in a bottom-up manner (because the Python recursion stack is limited).

Note that, assuming your subproblems from part (a) only find the size of the best result, you should also keep parent pointers so that you can reconstruct the actual subsequence.

Submit `progress.py` to the class website.

Part B: Due Thursday, May 7th

1. **(25 points)** Linear Rubberband Approximation

Assume an unknown function, $f(x)$, generated a sequence of n points on a plane. Using these n points, we now want to produce an approximation $g(x)$ to the unknown function $f(x)$ (at least for x in the range of the points given). We will do this using a piecewise linear function of at most k linear pieces, where k is a given integer. We construct g in the following manner:.

- Let each point, p_i be the pair of coordinates, (x_i, y_i); assume they are labeled in order of increasing x coordinate. Suppose from the n given points, we form a subset by taking $k+1$ of those points. We will label the points in the subset q_j such that $q_j = p_{i_j}$, for $1 \leq j \leq k+1$ and $1 = i_1 < i_2 < \cdots < i_{k+1} = n$.
- The piecewise function $g(x)$ then consists of the straight line segments connecting every two adjacent q points starting at q_1 and ending at q_{k+1} (there are k linear pieces).
- We define the approximation error from our piecewise linear function g of k pieces on the original point set by summing the squared errors between the n original points and the approximation $g(x)$:

$$error_k = \sum_{i=1}^{n} (y_i - g(x_i))^2$$

note: the approximation error is zero for the $k+1$ points p_{i_j}.

(a) **(20 points)** Assuming we are given the sequence of n points and an integer k, where $1 \leq k \leq n$, state the dynamic programming problem we need to solve in order to find the g that minimizes $error_k$. That is, state the subproblems we need to solve, the relation between the problems, and the base case(s).

(b) **(5 points)** To avoid overfitting, suppose we want to minimize:

$$h(k) = c * k + error_k$$

where c is some given constant. Describe a simple approach for doing so. Can you improve the running time if $h(k)$ is assumed to be convex?

2. **(25 points)** Viterbi's Algorithm

The Viterbi algorithm is used to estimate the most likely sequence of states given a sequence of observations. It is used in speech recognition, wifi networks, and many other applications. In this problem we develop a variant of Viterbi for a problem that is a bit easier to state than the standard one.

(a) **(10 points)** Consider a directed graph $G(V, E)$ with a special source vertex s and with a label $\ell(e)$ for each edge $e \in E$. The labels may repeat and several edges may have the same label. Given the graph and a sequence of labels $\ell_1, \ell_2, \ldots, \ell_k$, describe an algorithm that finds a path $s = v_0, v_1, v_2, \ldots, v_k$ such that $\ell_i = \ell(v_{i-1}, v_i)$. That is, the labels along the path should match the given sequence of labels. Analyze the running time of your algorithm.

(b) **(15 points)** Now suppose that each edge $e \in E$ also has a probability $p(e)$, such that the sum of probabilities of the edges that leave each vertex is 1. Extend your algorithm to output the *most probable* path consistent with a given sequence of labels, where the probability of a path is the product of edge probabilities along it. Analyze the running time of the algorithm.