Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — April 24, 2008
Professors Srini Devadas and Erik Demaine — Handout 12

Problem Set 6

This problem set is due **Thursday May 8** 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.

---

1. **(10 points)** Fibonacci

   We define the Fibonacci numbers as follows:

$$F_0 = 0$$
$$F_1 = 1$$
$$\forall n > 1, F_n = F_{n-1} + F_{n-2}$$

   For this problem, you will code four versions of `fib(n)`, all of which should return the same answers.

   Download `ps6_fib.zip`.

   (a) Write `fib_recursive(n)`. It should implement the recursion directly and take time exponential in n.

   (b) Write `fib_memoize(n)`. It should be recursive like `fib_recursive(n)`, but it should memoize its answers, so that it runs in time $O(n)$.

   (c) Write `fib_bottom_up(n)`. Instead of working top-down like in the previous two examples, start from the bottom, recording your results in a list. This code should also take $O(n)$ time.

   (d) Write `fib_in_place(n)`. It should work bottom-up like the previous example, but use only $O(1)$ space instead of accumulating answers into a list.

   Submit `fib.py` to the class website.

2. **(15 points)** Making Change

   You've just signed up as a software engineer for a new intergalactic trading post in Deep Space 6. For each transaction, you are given a list of coin denomination values, e.g., $denomination = [1, 5, 10, 17]$, and an amount of change, C. You have an unlimited number of each type of coin. Your goal is to find the shortest possible list of coins that adds up to C. For simplicity, assume that there is always a penny $1 \in denomination$ and that the desired change C is an integer, so the problem always has a solution.

   (a) Clearly state the set of subproblems that you will use to solve this problem.
   Solution: Let $dp[i]$ be the smallest number of coins that can be used to add up to value i.

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

   Solution:

$$dp[0] = 0$$

$$dp[i] = 1 + \min_{d \in denomination} dp[i - d]$$

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

   You should end up with a *pseudopolynomial* running time, meaning that the polynomial includes some power of C. This is not exactly the same as being polynomial with respect to the size of the input, because it only takes $\lg C$ bits of input to represent the number C.

   Solution: There are C problems, for each value between 0 and C. Each subproblem takes $O(k)$ time to solve, where k is the length of *denomination*. The running time is $O(k * C)$.

   (d) Download `ps6_change.zip`.

   Write a function `make_change(denomination, C)` which returns a list of coins that add up to C, where the size of the list is as small as possible. 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 `change.py` to the class website.

3. **(15 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. Thinking back to your 6.006 days, you have a vague recollection that longest increasing subsequence is one of those problems that can be solved by Dynamic Programming.

   (a) Clearly state the set of subproblems that you will use to solve this problem.
   Solution: Let $dp[i]$ be the length of the longest increasing subsequence ending at element i.

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

   Solution: Let A be the array of awesomeness scores.

   $$dp[0] = 0$$

   $$dp[i] = 1 + \max_{j<i, A[j]<A[i]} dp[j]$$

   Simply try all possible previous elements j. Check to see if that score was smaller. If so, we can use the best subsequence ending there, making it one longer because we also use element i.

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

   Solution: There are n subproblems, where n is the number of awesomeness scores. Each subproblem can be solved in time $O(n)$, for a total running time of $O(n^2)$.

   (d) Download `ps6_progress.zip`.

   Write a function `longest_increasing_subsequence(scores)` 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.

4. **(20 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.

Download `ps6_image.py`. You will also need 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.