Version 1.0 ***due:*** *Fri, Dec 6th @ 11:59pm*

1 Introduction

In this assignment you will be exposed to several dynamic programming (DP) concepts and come up with approaches for dealing with them in an abstract way. You will also provide an implementation of a dynamic programming algorithm known as seam carving. In the first half of this assignment, you probe several problems and elicit efficient dynamic programming solutions. You will also be asked to provide descriptions of the dynamic programming algorithms needed. In the second half of this assignment, you are asked to implement the seam carving algorithm, which has an interesting application for image resizing.

Note: Instructions for testing are at the end of this document.

2 Files

After downloading the assignment tarball from Autolab, extract the files from it by running `tar -xvf dplab-handout.tgz` from a terminal window. You should see the following files:

1. `Makefile`
2. `submit`
3. `support/`
4. `sources.cm`
5. `images/`
6. `* MkSeamFind.sml`

You should only modify the last file, denoted by `*`. Additionally, you should create a file called `written.pdf` which contains the answers to the written part of the assignment.

3 Submission

To submit your assignment: open a terminal, `cd` to the `dplab-handout` folder, and run `make`. Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "Handin your work" link.

4 Dynamic Programming

Here is a model solution to the "breaking a string into minimum number of palindromes" problem. Throughout this problem $S[i..j]$ denotes the substring between position i and position j. For example, if $S =$ `"ABC"`, then $S[1..2]$ is the string `"BC"`.

Question: Given a string S of length n, describe an $O(n^2)$ time dynamic programming problem to compute the minimum number of palindromes S can be broken into.

Answer: The subproblems are:

`DP[i]` is the minimum number of palindromes in the last i characters of S
`IS_PAL[i,j]` is true if $S[i..j]$ is a palindrome

The recurrences are:

$$\texttt{DP}[i] = 1 + \min_{j<i \text{ and } \texttt{IS\_PAL}[j+1,i]} \texttt{DP}[j]$$
$$\texttt{IS\_PAL}[i,j] = S[i] == S[j] \text{ andalso } \texttt{IS\_PAL}[i+1, j-1]$$

The base case is `DP[0] = 0` and `IS_PAL[i][i] = true` if $S[i] == S[i]$
The final answer is `DP`$[n]$

In calculating `DP`, there are n subproblems, each of which has $O(n)$ non-recursive work (looping over $O(n)$ choices of j doing constant non-recursive work for each j), so the amount of work in calculating `DP` this step is $O(n^2)$.
In calculating `IS_PAL`, there are $O(n^2)$ subproblems, each of which has $O(1)$ non-recursive work, so the total work in calculating `IS_PAL` is $O(n^2)$.
So the total amount of work is $O(n^2)$.

Writeups should include: The subproblems you are working with, how to get the final answer, any base cases, and recurrences for computing subproblems in terms of other subproblems—and of course, runtime analysis.

Writeups should *not* include: Correctness proofs.

Writeups should be short (excessively long or unclear answers will lose points even if they are correct).

4.1 #isnt15210thebest[1]

After finishing Rangelab, you post an overexcited tweet with the hashtag *#imanagedthelab*. Your friend, who is unfamiliar with Twitter, doesn't know how to insert spaces between the words in your hashtag to interpret it appropriately.

He first tries to greedily scan through until he first sees a whole word, insert a space, and then continue scanning. Depending on his dictionary, he might see "*I man age*" and then get stuck because no words start with "*dth*". He could then backtrack and try a longer first word, which would produce "*I'm a nag*" and then might also get stuck.

Unfortunately, any backtracking algorithm like this will be exponential in the worst case. Your friend is desperate to find out what exactly your hashtag means, so you need to give him a better algorithm for determining whether it is possible to insert spaces into a spaceless string to produce a sequence of valid English words.

Task 4.1 (10%). Give a $O(n^3)$ dynamic programming algorithm to solve this problem. Assume you have a constant-time function `isWord`(s) which looks the word s up in the dictionary and returns a boolean indicating whether it is a valid word.

4.2 The Frog Friend

After successfully helping your friend master hashtags, you yearn to take a well-deserved vacation with the Penelopean Flame-Princess. While on the road, you stop to drink from a nearby river when the Princess calls out: "Look `<insert name here>`! Look at how these tiny frog-people on either side of the bank have built their teeny villages!" You wipe your brow, and crouch down to stare intently at the small frogs below.

Neat! But wait a minute... every village on this bank has different colored frogs! And no village on the same side of the bank has frogs of the same color, but for every color, there are exactly two villages, each on a different side of the bank, that have frogs of that color. Pointing at a particularly bile-colored-village you murmur "Look how lonely and forlorn these slimy chaps look... we really should join them up with their kin from across the river! Let's build little froggy-bridges so that they could visit each other and be together with their like-colored froggy friends!"

The Flame-Princess, being more mathematically inclined, frames the question as such: You are given $2n$ villages as two sequences of x-coordinates, with n villages on each side of a river-bank. Each village on a particular bank takes on a unique color (which is shared with exactly one village on the opposing bank). You must build bridges between villages of the same color so that no two bridges cross each other. Given this constraint though, you want to help connect as many frogs as possible. That is, each bridge you can build has a weight (the number of frogs it would help), and you want to build the set of non-crossing bridges with the greatest possible weight.

Task 4.2 (10%). Describe an $O(n^2)$ dynamic programming algorithm to solve this problem.

[1] `http://bit.ly/1fx8SH3`

4.3 Jimmy Jiggling

Jimmy the Gorilla lives in a large jungle, where value is given to a local currency colloquially known as 'jimmies'. Having J nanabas (a rather tasty local fruit) but no jimmies, Jimmy decides to cash in, and convert his nanabas into jimmies (I mean, who wouldn't?). There are n denominations of jimmies with nanaba-values $j_1 < j_2 < ...j_n$ (Note that each denomination is in $\mathbb{Z}^+$).

Your goal is to come up with an algorithm that finds the fewest number of jimmies possible that suffice to generate value for J nanabas. If it is not possible to do so, you should be unrustled, detect this, and return an appropriate value.

Task 4.3 (10%). Describe a dynamic programming algorithm to solve this problem. What is the big-O complexity of your algorithm?

4.4 A Peculiar Pastime

Some of the gorillas in Jimmy's province, being enterprising entrepreneurs, have set up an entertaining new competition, and Jimmy, being heralded as the smartest and wittiest gorilla in the grand old jungle has just been challenged to compete! It's your job, as Jimmy's pal, to help him devise a fool-proof algorithm to win this game.

THE GAME

Jimmy is given a set of $k \geq 1$ alphabets $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ (k is not a constant). He has m rules of the form:

"*replace a single character x with yz*" (which we write $x \to yz$)

Here, $x, y, z \in \Sigma$ are characters from the alphabet set. At each timestep, for each character in your string thus far, he can either leave it alone or apply a rule to it. Several rules can be applied in the same timestep, and each rule can be applied multiple times.

For example, suppose the alphabet was $\Sigma = \{\sigma_1 = A, \sigma_2 = B, \sigma_3 = C\}$ and the rules were:

- **Rule 1**: $A \to BC$
- **Rule 2**: $B \to AC$
- **Rule 3**: $C \to AB$

This means that if we have the string *ACC*, we could apply Rule 3 on both occurrences of C in the same round, yielding *AABAB*. We could also get *BCABAB* by applying Rule 1 on A and Rule 3 on the two occurrences of C.

Now consider deriving the string *ACAB* from A. One possibility is to apply Rule 1 to get $A \to BC$—then, apply Rule 2 on B and Rule 3 on C to get *ACAB*. Alternatively, we could start with A and use Rule 1 to get *BC*; then, apply Rule 2 on B and leave C alone, so we get *ACC*. After that, apply Rule 3 on the last character gives us *ACAB*. In this case, the first path is shorter, hence a better path.

Task 4.4 (10%). Your task, being Jimmy's bosom buddy, is to compute, in $O(n^3mk)$ work, a dynamic programming algorithm to calculate the *minimum* number of steps it takes to make a string S of length

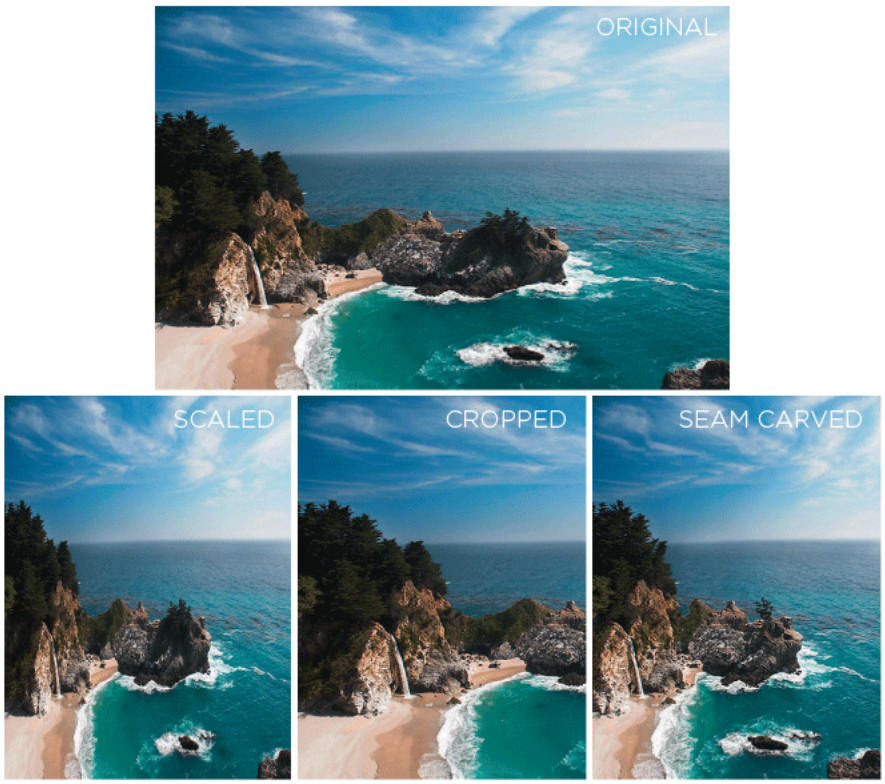
n from the string "σ_1". If you cannot make S from "σ_1" using the provided rules, your algorithm should detect this. For concreteness:

- The alphabets are numbers 1 through k
- The rules are given as a sequence of triples, i.e. the rule $x \to yz$ is represented as (x, y, z)
- The "target" string S is given as an `int seq`

Good luck and *godspeed*!

5 Seam Carving

Seam Carving is a relatively new technique discovered for "content-aware image resizing" (Avidan & Sheridan, 2007). Traditional image resizing techniques involve either scaling, which results in distortion, or cropping, which results in a very limited field of vision.



The technique involves repeatedly finding 'seams' of least resistance to add or to remove, rather than straight columns of pixels. In this way the salient parts of the image are unaffected. It's a very simple idea, but very powerful. You will work through some simple written problems and then implement the algorithm yourself to use on real images. For simplicity, we will only be dealing with horizontal resizing. To motivate this problem, watch the following SIGGRAPH 2007 presentation video which demonstrates some surprising and almost magical uses of this technique:

SIGGRAPH 2007 Seam Carving Presentation

5.1 Logistics

5.1.1 Representation

Images will be imported by a Python program that uses image libraries which may not be compatible with your local machine. Make sure to use one of the Andrew machines to run your program. The imported `image` type is defined as

```
type pixel = { r : real, g : real, b : real }
```

```
type image = { width : int, height : int, data : pixel seq seq }
type gradient = real
```

5.1.2 Gradients

To find a seam of least resistance, we define the following: for any two pixels $p = (r_1, g_1, b_1)$ and $q = (r_2, g_2, b_2)$, the *pixel difference* δ is the sum of the differences squared for each RGB value:

$$\delta(p,q) = (r_2 - r_1)^2 + (g_2 - g_1)^2 + (b_2 - b_1)^2$$

Then, for a given pixel $p_{i,j}$, the *gradient* $g(i,j)$ is defined as the square root of the sum of its difference from the pixel on the right and its difference from the pixel below it.

$$g(i,j) = \sqrt{\delta(p_{i,j}, p_{i,j+1}) + \delta(p_{i,j}, p_{i+1,j})}$$

This leaves the right-most column and bottom row undefined. For an image with height n and width m, we define $g(n-1,j) = 0.0$ for any j, and $g(i,m-1) = \infty$ for any i. Using the gradient as a cost function, we can now develop an algorithm to compute a lowest-cost seam for any image.

5.2 Specification

Task 5.1 (10%). Implement the function

```
generateGradients : image -> gradient seq seq
```

in `MkSeamFind.sml` which converts a raw `image` type into a 2D sequence of gradients $g(i,j)$ as defined above. Your implementation should have work in $O(nm)$ and constant span.

5.2.1 Seam Finding Algorithm

Let's work through a simple example. Suppose we have a 4x4 image with the following table of gradient values already computed (ignoring the right-most column of ∞'s and the bottom row of zeroes):

i ↓ j →	**0**	**1**	**2**	**3**
0	5	4	1	3
1	7	2	3	5
2	6	5	6	1
3	3	2	7	8

Task 5.2 (7%). A valid vertical seam must consist of m pixels if the image has height m, and each pixel must be *adjacent* in the sense that if the seam at row i is at column j, the seam at rows $i-1$ and $i+1$ are limited to columns $j-1$, j, or $j+1$. The cost of a seam S is $\sum_{p_{i,j} \in S} g(i,j)$. Let $m(i,j)$ be the minimum cost to get from row 0 to $p_{i,j}$. Fill in the table for $m(i,j)$.

i ↓ j →	**0**	**1**	**2**	**3**
0		4		
1	11			
2				
3				

Task 5.3 (3%). What is the lowest cost vertical seam in this image?

Task 5.4 (5%). For the general case, write the mathematical definition of $m(i, j)$.

Task 5.5 (25%). Using this insight, implement the function

```
findSeam : gradient seq seq -> int seq
```

in `MkSeamFind.sml` which finds the lowest-cost seam in the given image, where the seam is represented as an ordered sequence of column indices going from the top row of the image to the bottom row. Your implementation should have $\Theta(mn)$ work.

5.2.2 Testing

As usual, you will be required to test your code. But it's going to be a little more fun this time! In the handout directory we've given you a few sample images. After completing the above tasks, you should be able to perform the following in the REPL:

```
$ smlnj
Standard ML of New Jersey v110.73 [built: Mon Aug 29 23:55:00 2013]
- CM.make "sources.cm";
[autoloading]
...
[New bindings added.]
val it = true : bool
- Tester.removeSeamsFile ("images/sample.jpg", "images/my-sample-50.jpg", 50);
val it = () : unit
```

This takes the image `images/sample.jpg`, removes 50 seams from it, and stores the resulting image in `images/my-sample-50.jpg`. You should compare your results with the given `sample-50.jpg`, `sample-100.jpg`, and `sample-200.jpg` samples. For reference, our solution removes 100 seams from `cove.jpg` in < 10 seconds. Removing more seams from higher resolution images will naturally take longer.