Lab 9 - *DPLab*

Due: *Mon, Nov 24th, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Fall '14)

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 is an interesting application of image resizing.

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf dplab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `images/`
3. * `MkSeamFind.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `dplab` 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 DP solution to the "breaking a string into minimum number of palindromes" (MP) problem. Throughout this problem $S\langle i,j\rangle$ denotes the substring between position i and position j. For example, if $S =$ `"ABC"`, then $S\langle 1,2\rangle$ is the string `"BC"`.

Question:

Given a string S of length n, describe an $O(n^2)$ work and $O(n \log n)$ span dynamic programming algorithm to compute the minimum number of palindromes S can be broken into.

Answer:

Recursive solution:

```
fun MP(S) = let
  % Is the subtring $S\langle i,j\rangle$ a palindrome
  fun isPal($i,j$) =
    case ($i = j$) of
      true ⇒ true
    | false ⇒ ($S_i = S_j$) ∧ isPal($i+1, j-1$)

  % The minimum number of palindrones in first i characters of S
  fun MP'(0) = 0
    | MP'($i$) = $1 + \min\{$MP'($j$) : $j \in \{0,\ldots,i-1\}$ | isPal($j, i-1$)$\}$
in
  MP'($|S|$)
end
```

Sharing:
For $n = |S|$ there are at most n^2 distinct calls to `isPal`(i,j) since i and j range from 0 to $n-1$, and $n+1$ distinct calls to `MP'`(i) since i ranges from 0 to n.

DAG and costs:
The DAG for calculating all the `isPal`(i,j) has a total of $O(n^2)$ nodes, has depth n, and each node has constant work and span. The work is therefore $O(n^2)$ and the span is $O(n)$.

The DAG for `MP'`(n) has n nodes, has depth n, and each node has $O(n)$ work and $O(\log n)$ span (for the min). The DAG therefore has work $O(n^2)$ and span $O(n \log n)$.

Both these costs are within the bounds.

Keep in mind that your responses should be short (excessively long or unclear answers will lose points even if they are correct), and not contain correctness proofs. To reiterate, as in the example above, each response should include the following 3 components:

1. A recursive solution clearly indicating the subproblems you are solving: `isPal`(i,j) and `MP'`(i) in the example.
2. Justification of how the subproblems are shared.
3. A description of the DAG(s) along with the work and span of the DAG(s).

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 $O(n^2)$ work and $O(n \log n)$ span dynamic programming algorithm to solve this problem. Assume you have a constant-work 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!"

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.

Task 4.2 (10%). Describe an $O(n^2)$ work, $O(n \log n)$ span dynamic programming algorithm that calculates the *maximum* number of bridges that you can build to help the frogs.

4.3 Freddy and Friends

Freddy the frog is playing a fun game with his froggy friends. The game is called *leapfrog* and it works like this: There are n lilypads on the river, each one at a distinct, nonnegative integer number of meters from the start of the river. When it's a frog's turn, they pick a lilypad to start on, and hop down the river

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

(they can only go down the river). The goal is to land on as many lilypads as possible. What's the catch? Frogs have terrible balance. Specifically, if a frog's first hop is k meters long, every hop that frog makes must be exactly k meters. So, in the example below, the best sequence of hops begins at position 3 and makes every hop of size 3:

$$\langle 1, 2, 3, 4, 6, 9, 12, 14, 15 \rangle$$

Task 4.3 (15%). Help Freddy out and figure out the best move he can make (the best start and hop size)! Your solution should be an $O(n^2)$ work dynamic programming algorithm, where n is the number of lilypads on the river.

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 an *alphabet* Σ of $k \geq 1$ *characters*, $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, and m rules that apply to the characters. Each rule has the form:

"*Replace a single character* σ_x *with* $\sigma_y \sigma_z$" (which we denote $\sigma_x \to \sigma_y \sigma_z$)

Where $\sigma_x, \sigma_y, \sigma_z \in \Sigma$, and note that k and m are not constants. At each time step of the game, for each character in the string thus far, Jimmy can either choose to leave it (the character) alone, or apply a rule to it. Several rules can be applied in the same time step, 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**: $C \to AB$

Then 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, starting with A, apply Rule 1 to get BC, then apply Rule 2 on B and leave C alone to get ACC. Finally, applying Rule 3 on the last character gives us $ACAB$, as desired. In this case, the first path is shorter, and is thus considered to be a *better* path.

Task 4.4 (20%). Your task, being Jimmy's bosom buddy, is to come up with a $O(n^3 mk)$ work dynamic programming algorithm to calculate the *minimum* number of steps it takes to make a string S of length n, starting from "σ_1" (i.e. starting from the singleton string containing the first character in the alphabet). If S cannot be formed from "σ_1" using the provided rules, your algorithm should detect this.

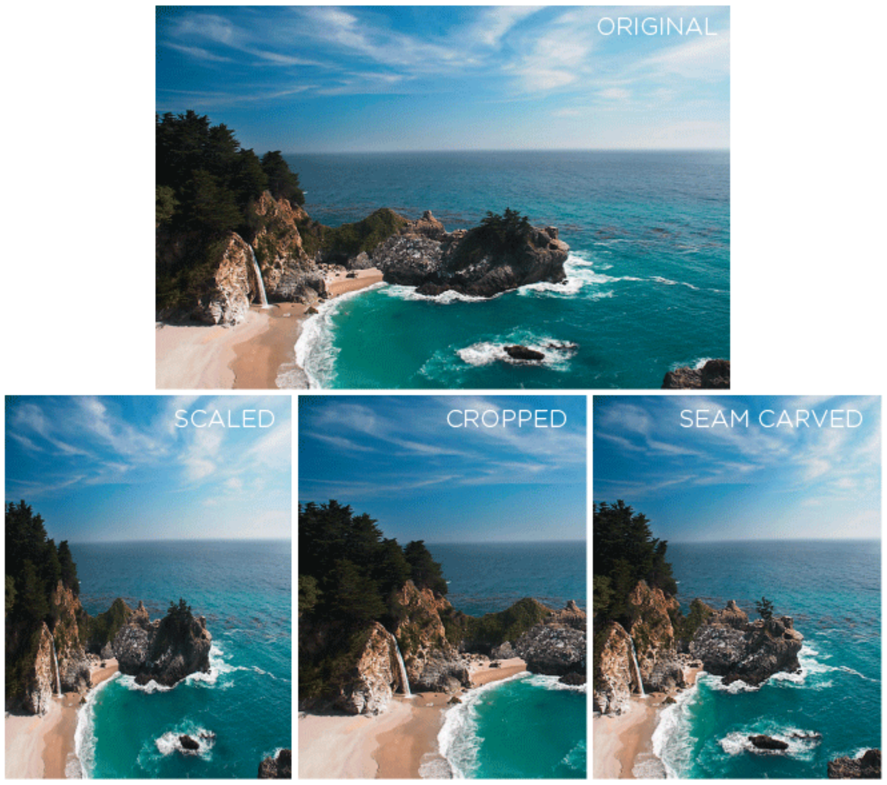
For concreteness:

- The alphabet Σ contains the integers 1 through k
- The *target* string S is given as a sequence of integers
- The rules are given as a sequence of triples (i.e. the rule $\sigma_x \to \sigma_y \sigma_z$ is represented as $(\sigma_x, \sigma_y, \sigma_z)$)

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:

`http://www.youtube.com/watch?v=6NcIJXTlugc`

5.1 Pixels and Gradients

Images will be imported by a Python program that uses some image libraries which may not be compatible with your local machine, so make sure to use one of the Andrew machines for the duration of this lab. These imported images are represented by the following SML types:

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

where each `pixel` is represented by its r (red), g (green), and b (blue) compositions, and an `image` encapsulates its dimensions as well as a 2D-array of pixels (`pixel seq seq`). Also, recall the SML *record* type (`https://piazza.com/class/hp3ascm5zmb49c?cid=8`).

Central to the Seam Carving algorithm is finding a seam of *least resistance*. To help us with this, for any two pixels $p_1 = (r_1, g_1, b_1)$ and $p_2 = (r_2, g_2, b_2)$, we define their *pixel difference* $\delta(p_1, p_2)$ to be the sum of the differences squared for each RGB value:

$$\delta(p_1, p_2) = (r_2 - r_1)^2 + (g_2 - g_1)^2 + (b_2 - b_1)^2$$

Then, for a given pixel $p_{i,j} = I[i, j]$ (i.e. the pixel at row i and column j of image I), the *gradient* $g(i, j)$ is defined as the square root of the sum of its *pixel difference* with the pixel on the right and its *pixel difference* with the pixel below it:

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

Note that 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 given image.

5.2 Seam Finding Algorithm

To begin, let us work through a simple example. Suppose we have a 4×4 image with the following table of gradient values already computed. In this example, we ignore the convention of have a right-most column of ∞s and bottom-most row of 0s.

$i \downarrow j \rightarrow$	**0**	**1**	**2**	**3**
0	5	4	1	3
1	7	2	3	5
2	6	5	6	1
3	3	2	7	8

A valid vertical seam S 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, then the seam at rows $i-1$ and $i+1$ must be from columns $j-1$, j, or $j+1$. The cost of the seam C_S is:

$$C_S = \sum_{p_{i,j} \in S} g(i, j)$$

Let M be the table below, and let G be the table above. We define each $M(i, j)$ to be the minimum cost to get from $G(0, y)$ to $G(i, j)$, where the starting column y must respect the *adjacency* rule above. **Note:** Make sure you understand how each cell of table M was derived.

$i \downarrow j \rightarrow$	**0**	**1**	**2**	**3**
0	5	4	1	3
1	11	3	4	6
2	9	8	9	5
3	11	10	12	13

The vertical seam with the lowest cost in this example is $S = \langle 2, 1, 1, 1 \rangle$, uniquely defined by its column indices, going from top to bottom, and its cost is $C_S = 10$.

Task 5.1 (45%). Using this insight, in `MkSeamFind.sml`, implement the function

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

which finds the lowest-cost seam in a given image, **which is preprocessed into its gradient values**, as described in Section **??**. The resulting seam is represented as an ordered sequence of column indices going from the top row of the image to the bottom row.

For full credit, your implementation must have $O(mn)$ work and span. **Hint:** it would be useful for you to think about what the mathematical definition of $M(i, j)$ is, using the above example.

As a sanity check, our implementation has under 30 lines of code.

5.3 Testing

As usual, you will be required to test your code. But it's going to be a little more fun this time! We've given you a few sample images in the `images/` directory with which you can run your Seam Carving implementation on. After completing the above tasks, you should be able to perform the following in the terminal:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
...
- Tester.removeSeamsFile("../images/cove.jpg","../images/my-cove-50.jpg",50);
```

where `Tester.removeSeamsFile` (P_i, P_o, n) takes the image at path P_i, removes n seams from it, and stores the resulting image at path P_o. You should compare your results with the given `sample-50.jpg`, `sample-100.jpg`, and `sample-200.jpg` files in `images/`.

For reference, our solution removes 100 seams from `images/cove.jpg` in $<$ 10 seconds. Removing more seams from higher-resolution images will naturally take longer.

Our private tests on Autolab have a wide range of different kinds of images, from plain, flat colors to images with busy landscapes (one of them is called "TimesSquare"... go figure!), and images with human faces in them. You are encouraged to test locally on all these different kinds of images to see how well your Seam Carver works!

Good luck and ***have fun***!