600.363/463 Algorithms - Fall 2013
Solution to Assignment 4

(30+20 points)

I (10 points)

This problem brings in an extra constraint on the optimal binary search tree - for every node v, the number of nodes in both of its subtrees should be at least 1/5 of the number of nodes in the tree rooted at v. Therefore in searching for the optimal root of a subtree ranging from k_i to k_j, instead of taking all nodes as candidate roots, we must constrain our search within the middle 3/5 of the nodes. Let k_r be the root, then left subtree contains $2r - 2i + 1$ nodes:

$$\{k_i, k_{i+1}, \cdots, k_{r-1}, d_{i-1}, \cdots, d_{r-1}\}$$

and the right subtree contains $2j - 2r + 1$ nodes:

$$\{k_{r+1}, k_{r+2}, \cdots, k_j, d_r, \cdots, d_j\}.$$

Note that there are $(j - i + 1) * 2 + 1 = 2j - 2i + 3$ nodes from k_i to k_j. Let

$$\begin{cases} 2r - 2i + 1 \geq & \frac{1}{5}(2j - 2i + 3) \\ 2j - 2r + 1 \geq & \frac{1}{5}(2j - 2i + 3) \end{cases}$$

Solving the above inequalities,

$$\begin{cases} r \geq & \frac{1}{5}(4i + j - 1) \triangleq r_1 \\ r \leq & \frac{1}{5}(i + 4j + 1) \triangleq r_2 \end{cases}$$

The resulting recursion is:

$$e[i, j] = min_{r_1 \leq r \leq r_2} \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j. \end{cases}$$

where

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{i-1}^{j} q_l.$$

The algorithm is shown below.

Algorithm 1: Optimal-Weight-Balanced-BST(p,q,n)

```
for i ← 1 to n do
    e[i, i − 1] ← q_{i−1};
    w[i, i − 1] ← q_{i−1};
end
for l ← 1 to n do
    for i ← 1 to n − l + 1 do
        j ← i + l − 1;
        e[i, j] ← ∞;
        w[i, j] ← w[i, j − 1] + p_j + q_j;
        ▷ Here the search is to the middle 3/5 of he nodes.
        r_1 = ⌊(4i + j − 1)/5⌋ + 1;
        r_2 = ⌈(i + 4j + 1)/5⌉ − 1;
        if r_1 ≤ r_2 then
            for r ← r_1 to r_2 do
                t ← e[i, r − 1] + e[r + 1, j] + w[i, j];
                if t < e[i, j] then
                    e[i, j] ← t;
                    root[i, j] ← r;
                end
            end
        end
    end
end
return e and root ;
```

The three nested loops take no more than n values for each loop variable, thus the algorithm takes $O(n^3)$ time.

II (10 points)
(The key point to this problem is allowing a batch of operations consisting of enqueue/dequeue and shuffle). The simulation is done in phases. At the beginning of a phase if the deque contains elements $a_1, a_2, \cdots, a_k$ then we store $a_{k/2}, a_{k/2-1}, \cdots, a_1$ in locations $1, 3, 5, \cdots$ of the array and $a_{k/2+1}, a_{k/2+2}, \cdots, a_k$ in locations $2, 4, 6, \cdots$. There are two pointers, one points to the left end a_1 and the other points to the right end a_k. The phase consists of simulationinghe next $k/2$ dequeue operations in a straightforward way. Then if the deque length becomes $k'(\leq k)$, it gets split in halves and stored in the above way. The $k/2$steps of dequeue takes $2(k/2) = k$ RAM steps for simulation and at most $2k'(\leq 3k)$ steps for redistribution. Hence it takes no more than $\frac{k+3k}{k/2} = 8$ steps per step of deque.

III (10 points)
Instead of using the length as the objective function, we redefine the objective function as the weighted value of the common subsequence. Let $c[i, j, a]$ be the maximum total weight among all the common subsequence of $x_1x_2\cdots x_i$ and $y_1y_2\cdots y_j$ that ends with symbol a. Then we define the $prev_x(i, a)$ and $prev_y(j, a)$ as the biggest index of the subsequence of $x_1\cdots x_i$ and

$y_1 \cdots y_j$ ended with a respectively:

$$prev_x(i,a) = \begin{cases} \max\{l\} \text{ such that } x_l = a & \text{if there exists such an } a \\ 0 & \text{otherwise} \end{cases}$$

$$prev_y(j,a) = \begin{cases} \max\{l\} \text{ such that } y_l = a & \text{if there exists such an } a \\ 0 & \text{otherwise} \end{cases}$$

Let $U = c[i-1, j, a]$ and $V = c[i, j-1, a]$. Hence the recursion formula becomes:

$$c[i,j,a] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \max\{U, V\} & \text{if } i,j > 0 \text{ and } a \notin \{x_i, y_j\}, \\ \max_b\{U, V, c[i-1, prev_y(j,a) - 1, b] + \delta(b,a)\} & \text{if } i,j > 0 \text{ and } a = x_i \text{ and } a \neq y_j \\ \max_b\{U, V, c[prev_x(i,a) - 1, j-1, b] + \delta(b,a)\} & \text{if } i,j > 0 \text{ and } a = y_j \text{ and } a \neq x_i \\ \max_b\{U, V, c[i-1, j-1, b]\} + \delta(b,a) & \text{if } i,j > 0 \text{ and } a = y_j = x_i \end{cases}$$

The maximum weight will be given by $max_a\{c[m,n,a]\}$. Let S be the dictionary of alphabets and s be its size. Note the maximization over b only happens in limited cases, the algorithm runs in $O(mns)$.
(Remarks: The third case is needed to consider all in which x_i matches up with some symbols in y. When it does match up we can move it up to the maximum position in y. The fourth case is similar. if you have not realized that you can limit to just the previous occurrence of the symbol a, then the third case will become

$$\max_{b,l}\{U, V, c[i-1, l, b] + \delta(b,a)\} \text{ if } a = x_i, a \neq y_j \text{ and } x_{l+1} = a$$

The fourth case is analogues similarly. Then the running time will be $O(smn \max\{m,n\}))$.

The psudo-code is given below.

Algorithm 2: LCS-MaxWeight(X, Y, δ)

```
m ← length(X);
n ← length(Y);
for a ∈ S do
    for i ← 0 to m do
        prev_x(i, a) = 0;
        c[i, 0, a] = 0;
        if x_{i−1} == a then prev_x(i, a) = i − 1;
        else prev_x(i, a) = prev_x(i − 1, a);
    for j ← 1 to n do
        prev_y(j, a) = 0;
        c[0, j, a] = 0;
        if y_{j−1} == a then prev_y(j, a) = j − 1;
        else prev_y(j, a) = prev_y(j − 1, a);
for i ← 1 to m do
    for j ← 1 to n do
        for a ∈ S do
            U = c[i − 1, j, a];
            V = c[i, j − 1, a];
            if x_i == y_j == a then
                c[i, j, a] =LOCAL-MAX(U, V, i − 1, j − 1, a);
            else if x_i == a&y_j ≠ a then
                c[i, j, a] =LOCAL-MAX(U, V, i − 1, prev_y(j, a) − 1, a);
            else if a == y_j&a ≠ x_i then
                c[i, j, a] =LOCAL-MAX(U, V, prev_x(i, a) − 1, j − 1, a);
            else
                if U > V
                then c[i, j, a] ← U;
                else c[i, j, a] ← V;
return c;
```

Algorithm 3: LOCAL-MAX$(U, V, i, j, a))$

```
weight ← −∞;
for b ∈ S do
    w ← c[i, j, b];
    if w > weight then weight = w; sym = b;
weight ← weight + δ(sym, a);
B[i, j, b] = " ↑ ";
if U > weight then weight = U;
if V > weight then weight = V;
return weight;
```

IV (bonus 10 points)

For any $1 \le d \le n$, define $v[i,j,d]$ as the minimum cost of the subtree containing keys $k_i, k_{i+1}, \cdots, k_j$ rooted at depth d, i.e.

$$e[i,j,d] = \min \sum_{l=i}^{j} p_l(d_l + 1 + d)^2$$

If k_r is the root of an optimal subtree containing keys $k_i, \cdots, k_j$ rooted at depth d, we have

$$\begin{aligned} e[i,j,d] &= \min_{i \le r \le j} \left\{ \sum_{l=i}^{r-1} p_l(d_l + 1 + d)^2 + \sum_{l=r+1}^{j} p_l(d_l + 1 + d)^2 + p_r(d_r + 1 + d)^2 \right\} \\ &= \min_{i \le r \le j} \left\{ \sum_{l=i}^{r-1} p_l(d'_l + 1 + 1 + d)^2 + \sum_{l=r+1}^{j} p_l(d'_l + 1 + 1 + d)^2 \right\} + p_r(0 + 1 + d)^2 \\ &= \min_{i \le r \le j} \{e[i, r-1, d+1] + e[r+1, j, d+1]\} + p_r(d+1)^2 \end{aligned}$$

and note that when $i = j$,

$$e[i,i,d] = p_i(d+1)^2$$

The algorithm is shown below.

Algorithm 4: Optimal-Weight-BST(p,q,n)

```
1  for d ← 1 to n do
2  |  for i ← 1 to n do
3  |  |  e[i,i,d] = p_i(d+1)^2;
4  |  end
5  end
6  for l ← 2 to n do
7  |  for i ← 1 to n − l + 1 do
8  |  |  j ← i + l − 1;
9  |  |  for d ← 1 to n do
10 |  |  |  e[i,j,d] = +∞;
11 |  |  |  for r ← i to j do
12 |  |  |  |  t ← e[i,r−1,d+1] + e[r+1,j,d+1] + p_r(d+1)^2;
13 |  |  |  |  if t < e[i,j,d] then
14 |  |  |  |  |  e[i,j,d] ← t;
15 |  |  |  |  |  root[i,j] ← r;
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 end
21 return e and root ;
```

Filling the table $e[i,j,d]$ takes $O(n^3)$ time, and search for the optimal root takes $O(n)$ time, hence the algorithm takes $O(n^4)$ time.

V (bonus 10 points)

One way to implement such a queue is using linked-list. There are two linked-lists: a data lined-list Q and a garbage linked-list G saving the released space. Each element in both Q and G has two pointers, one points to the next element and the other points to the previous. Assuming at the beginning the queue Q has k elements $a_1, a_2, \cdots, a_k$. The *head* pointers of Q points to a_1, and the *tail* pointer points to a_k. There is another flag Max recording the maximum allocated RAM space.

In the dequeue operation, a_1 is deleted from the data linked-list D by moving the *head* pointer to a_2 and the garbage linked-list G insert the space that element a_1 used to its end. This operation can be done straightforwardly. Suppose k' steps of dequeue is simulated, and k' is no more than k. After the k' steps, no new RAM space is required and no shuffle is needed. Therefore each dequeue step takes $k/k = O(1)$ RAM steps, and the space used is $3k$ spaces plus some constant spaces.

In the enqueue operation, if G is not empty, take a space from G and assign it to the data liked-list D by modifying the pointers. The length of G decreased by 1, while the length of D increased by 1, and the total allocated RAM is the same. If G is empty, then increase Max by 1 and attach it to D. Suppose k' steps of enqueue is simulated, where the spaces in G is used up and Max is increased afterwards. After the k' steps, the length of D is $k' + k$ and length of G is 0, and no shuffle is required. Therefore each enqueue step takes $k'/k' = O(1)$ RAM steps and $3(k + k')$ spaces.

The above mechanism ensures that the new space is allocated if and only if all the previously-allocated spaces are in use. Therefore the size of space is bounded by the size of the data, i.e. $O(S(n))$. Since no shuffle is required, and each step involves a constant number of operations, i.e. it performs $O(1)$ steps per step.