**Algorithm Qual Exam (Spring 2016)**
Closed book exam

Question 1 (15%)

In Java, it is known that the concatenation of Java's String objects is inefficient, while concatenation of Java's StringBuilder objects is more efficient.

Suppose for an application, we are only limited to concatenatation of a string with a letter; that is, no concatenation of two general strings. We want to create our own string class, called `myString`, to support efficient implementation of this concatenation operation.

Specifically, you need to support the following:

- `public myString( myString s, char c )`
  a constructor that returns the concatenation of string `s` and character `c`.
- `public String toString()`
  an instance method that returns a Java's String object with the same string content.

It is required that the constructor takes O(1) time, while `toString` takes time linear in the length of the string.

Give the Java codes for `myString` class.

If you are not familiar with Java, you may give the codes in C/C++/Python.

(Note: You will not be penalized for syntax errors. Just pay attention to the logic behind the design. You can also introduce helper methods that support your implememtation.)

**Answer:**

```
public class myString {
   private char c;
   private myString next;

   public myString(myString s, char ch) {
      c = ch;
      next = s;
   }

   public String toString() {
      if (next == null) return "" + c;
      return next.toString() + c;
   }
}
```

Question 2

Given a sequence $X = [x_1, x_2, \ldots, x_m]$, another sequence $Y = [y_1, y_2, \ldots, y_n]$ is a subsequence of $X$ if there exists a strictly increasing sequence $[i_1, i_2, \ldots, i_k]$ of indices of $X$ such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = y_j$.

In the longest common subsequence problem, given two sequences $X = [x_1, x_2, \ldots, x_m]$ and $Y = [y_1, y_2, \ldots, y_n]$, we want to return a maximum-length common subsequence of $X$ and $Y$.

(a) (15%) Design a dynamic programming solution that runs in $O(mn)$ time and space. Note: you are required to return the longest common subsequence, not just the length of the longest common subsequence.

**Answer:**

$$(b[i,j], c[i,j]) = \begin{cases} (\text{—}, 0) & \text{if } i = 0 \text{ or } j = 0 \\ (\nwarrow, c[i-1, j-1] + 1) & \text{if } i, j > 0 \text{ and } x_i = y_j \\ (\uparrow\ , c[i-1, j]) & \text{if } c[i-1, j] \geq c[i, j-1] \\ (\leftarrow, c[i, j-1]) & \text{otherwise} \end{cases}$$

From $b[m, n]$, we can retrace the $b$ table to reconstruct the longest common subsequence.

(b) (10%) Suppose we are only required to compute the length of the longest common subsequence. Improve your solution to run in space $O(\min(m, n))$ and time $O(mn)$.

**Answer:**

If the $c$ table is computed in a row-major order, we only need to keep track of the last row of $c$ entries. Thus, space usage is $O(m)$. Similarly, computing in a column-major order uses $O(n)$ space. By choosing the more favorable method, space used can be $O(\min(m, n))$.

(c) (15%) Now we want to incorporate into the solution for part (b) the return of the longest common subsequence. Explain how it can be done in space $O(k \cdot \min(m, n))$ and time $O(mn)$, where $k$ is the length of the longest common subsequence. Hint: make use of `myString` class (Question 1) with adaptation.

**Answer:**

We modify the $b[i, j]$ entries to hold the longest common subsequence of $X[1..i]$ and $Y[1..j]$.

$$b[i,j] = \begin{cases} \texttt{myString}(b[i-1, j-1], x_i) & \text{if } i, j > 0 \text{ and } x_i = y_j \\ b[i-1, j] & \text{if } c[i-1, j] \geq c[i, j-1] \\ b[i, j-1] & \text{otherwise} \end{cases}$$

Each $b[i, j]$ entry is a string of length $O(k)$. As in the solution for part (b), we only need to keep track of the last row or column of $b$ array. By also incoporating garbage collection using reference counters for reclaiming unused `myString` objects, the space used is $O(k \cdot \min(m, n))$. The overhead time for garbage collection is linear in the number of times the constructors for `myString` is invoked, which is $O(mn)$.

Question 3.

Consider a divide-and-conquer algorithm for matrix computations. An example is Strassen's method for matrix multiplication. We assume that the submatrices passed in recursive calls are the four quadrants of an $n \times n$ matrix, where $n$ is a power of 2.

Consider the following $8 \times 8$ matrix $A$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \end{bmatrix}$$

In memory, the matrix entries are organized in row-major order as in: 1 2 3 4 5 6 7 8 9 10 11 ... 64. That is, the entries of each quadrant of the matrix are not located in physically contiguous locations in memory. When passing a submatrix in recursive call, we cannot simply pass (by reference) the starting memory location of the submatrix. Instead, we need to pass by copying. Repeatedly passing submatrices in recursive calls by copying may not be efficient. One idea is to reorder the way the matrix entries are organized in memory so that entries that belong to the same quadrant are located together physically. The reordering is done by a recursive algorithm. To reorder the above $8 \times 8$ matrix $A$, we reorder submatrix $A_{11}$, followed by the reorder of $A_{12}$, then the reorder of $A_{21}$, and finally the reorder of $A_{22}$, where $A_{11} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \\ 17 & 18 & 19 & 20 \\ 25 & 26 & 27 & 28 \end{bmatrix}$, $A_{12} = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 13 & 14 & 15 & 16 \\ 21 & 22 & 23 & 24 \\ 29 & 30 & 31 & 32 \end{bmatrix}$, $A_{21} = \begin{bmatrix} 33 & 34 & 35 & 36 \\ 41 & 42 & 43 & 44 \\ 49 & 50 & 51 & 52 \\ 57 & 58 & 59 & 60 \end{bmatrix}$, and $A_{22} = \begin{bmatrix} 37 & 38 & 39 & 40 \\ 45 & 46 & 47 & 48 \\ 53 & 54 & 55 & 56 \\ 61 & 62 & 63 & 64 \end{bmatrix}$.

Recursively applying the reordering logic, matrix $A$ is represented in memory as follows:

1 2 9 10 3 4 11 12 17 18 25 26 19 20 27 28 5 6 13 14 7 8 15 16 21 22 29 30 23 24 31 32 33 34 41 42 35 36 43 44 49 50 57 58 51 52 59 60 37 38 45 46 39 40 47 48 53 54 61 62 55 56 63 64.

(a) (15%) Give Java/C/C++/Python codes that implement the reorder logic.

**Answer:**

```
reorderHelper(int a[][], int i, int j, int d, int b[], int k) {
  if (d==1) {b[k] = a[i][j]; return;}
  reorderHelper(a,   i,    j,       d/2,    b, k        );
  reorderHelper(a,   i,    j+d/2,   d/2,    b, k+  d*d/4);
  reorderHelper(a,   i+d/2,j,       d/2,    b, k+  d*d/2);
  reorderHelper(a,   i+d/2,j+d/2,   d/2,    b, k+3*d*d/4);
}

reorder( int a[][], int b[] ) { // a is of dimension n x n, b is an array of length n*n
  reorderHelper( a, 0, 0, n, b, 0 );
}
```

(b) (10%) Analyze the running time of the reorder algorithm in $\Theta$ notation in terms of $n$.

**Answer:**

$T(n) = 4T(n/2) + \Theta(1)$

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$ by master theorem as $\log_b a = \log_2 4 = 2 > 0 = d$.

(c) (10%) Assuming that your logic for part (a) is recursive, analyze (i) the depth of recursions, and (ii) the total number of recursive calls in terms of $n$ that are needed for reordering the entries of an $n \times n$ matrix. Note: your answer should **not** be in $\Theta$ or $O$ notation as we want to understand more accurately the efficiency of the conversion algorithm.

**Answer:**

(i) The depth of recursion is $\lg n$ where $n$ is the matrix dimension.

(ii) Total number of function calls is $1 + 4 + 4^2 + ... + 4^{\lg n} = \frac{4^{1+\lg n}-1}{4-1} \approx \frac{4^{1+\lg n}}{3} = \frac{4}{3}n^2$.

(d) (10%) Give Java/C/C++/Python codes that convert a matrix with reordered entries back to a matrix with entries in the usual order.

**Answer:**

```
reorderRevHelper(int a[][], int i, int j, int d, int b[], int k) {
  if (d==1) {a[i][j] = b[k]; return;}
  reorderRevHelper(a,   i,    j,       d/2,    b, k        );
  reorderRevHelper(a,   i,    j+d/2,   d/2,    b, k+  d*d/4);
  reorderRevHelper(a,   i+d/2,j,       d/2,    b, k+  d*d/2);
  reorderRevHelper(a,   i+d/2,j+d/2,   d/2,    b, k+3*d*d/4);
}

reorderRev( int a[][], int b[] ) { // a is of dimension n x n, b is an array of length n*n
  reorderRevHelper( a, 0, 0, n, b, 0 );
}
```