

SECTION A (3 QUESTIONS, 90 MARKS)

Use a separate book for Section A

1. (30 marks in total.)

This question concerns the class `QuadTree` discussed in lectures. The header file for `QuadTree` is given at the end of this question. Recall that a `QuadTree` represents a square image by dividing it (recursively) into four equal quadrants, with each quadrant corresponding to a subtree; the top-left, bottom-left, top-right and bottom-right quadrants in a division correspond (from left to right) to the 0'th, 1'st, 2'nd and 3'rd subtrees respectively.

Consider the 4 x 4 image containing 16 pixels.

1	2	3	4
5	6	6	7
8	6	6	9
10	11	12	13

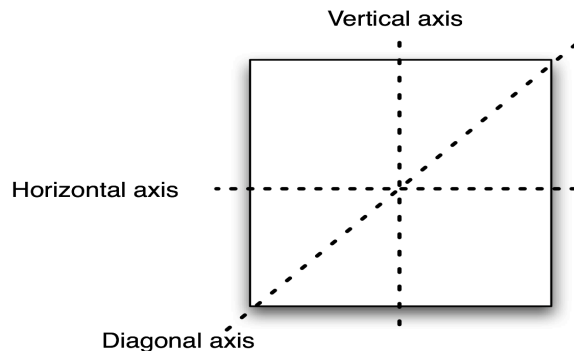
- (6 marks) Draw the `QuadTree` representation for this image.
- (2 marks) Draw the `QuadTree` representation for the image reflected around the vertical axis. (Use the diagram below to check the axis.)
- (2 marks) Draw the `QuadTree` representation for the image reflected around the horizontal axis. (Use the diagram below to check the axis.)
- (10 marks) Implement the following function specifications. Use a recursive implementation.

```
void recVertical(ptrNode t);  
// POST: Recursively flips the QuadTree t about the vertical axis
```

```
void recHorizontal(ptrNode t);  
// POST: Recursively flips the QuadTree t about the horizontal axis
```

- (2 marks) Starting with the `QuadTree` Root representing the 4 x 4 image above, draw the resulting 4x4 image which is represented by the tree after executing first the program `recVertical(Root)` and then `recHorizontal(Root)`.
- (8 marks) Now consider the diagonal axis depicted below. Implement a function which flips the image about the diagonal. (Use the diagram below to check the axis.)

```
void recDiagonal(ptrNode t);  
// POST: Recursively flips the QuadTree t about the diagonal axis
```



```

// Header file for the class QuadTree, containing declarations of the data structures
// and the constructors.

struct pixel {
    int xCoord;
    int yCoord;
};

struct QuadNode {
    int intensity;
    QuadNode* links[4];
};

typedef QuadNode* ptrNode;

class QuadTree
{
public:
    // Constructors

    QuadTree(istream& in, int N);
    // Constructor.
    // A compressed Quad Tree is built representing the image from file in,
    // Root is set to the root node, zm is set to the root node, and
    // path is set to the empty string.

private :

    ptrNode Root; // The pointer to the root: this corresponds to
                  // the whole image.
};

```

2. (30 marks in total.)

This question is related to the length of the longest common subsequence algorithm studied in lectures and tutorials. Recall that a subsequence of s is obtained by deleting any number of elements from any position in s . The code is reproduced below, and it assumes a table `LookUp` of integers and two strings `s1` and `s2`. The algorithm iteratively computes the table so that `LookUp[h][v]` is the length of the longest common subsequence of `s1.substr(h,N)` and `s2.substr(v,M)`.

```
int lcsc(int i, int j) {
// PRE: i <= s1.length()==N; j <= s2.length()==M;
// POST: returns the length of the longest common subsequence
//       of s1.substr(i,N) and s2.substr(j,M).
  if (LookUp[i][j] == -1) { // If LookUp[i][j] is undefined, compute it ...
    for (int h= s1.length(); h >= i; h--) {
      for (int v= s2.length(); v >= j; v--) {
        if(h >= s1.length() || v >= s2.length()) LookUp[h][v]= 0;
        else if (s1[h] == s2[v]) LookUp[h][v]= 1 + LookUp[h+1][v+1];
        else
          LookUp[h][v]= max(LookUp[h+1][v],
                             LookUp[h][v+1]);
      }
    }
  }
  return LookUp[i][j];
}
```

The function `max(m,n)` returns the maximum of m and n . For string s , `s.length()` is the length of s , and when $N == s.length()$, `s.substr(i, N)` is the substring of s consisting of the characters from index i to the end.

- (a) (7 marks) Construct the `LookUp` table for the strings `s1 == "carrot"` and `s2 == "article"`. Using your table, write down `lcsc(0,0)`; write down `lcsc(2, 0)`.

A wordprocessor allows strings ss to be edited in the following way:

- Any character in ss may be **deleted**;
- Any character may be **inserted** into any position of ss .

Given two strings $s1$ and $s2$ the *minimum edit distance* between $s1$ and $s2$ is defined to be the *smallest* number of insertions and deletions required to transform $s1$ into $s2$. For example if `s1 == "hart"` and `s2 == "rate"`, then the minimum edit distance is 4 since two letters must be deleted from `"hart"` to make `"rt"` and then two letters inserted into that to make `"rate"`.

- (b) (2 marks) Compute the minimum edit distance between `s1 == "carrot"` and `s2 == "article"`.
- (c) (2 marks) If `s1 == "hapn"` and `s2 == "happen"` how many characters need to be inserted into $s1$ to transform it into $s2$? How many characters need to be deleted?
- (d) (2 marks) If $s1$ is a subsequence of $s2$ write down, in terms of `s1.length()` and `s2.length()` the minimum edit distance to transform $s1$ into $s2$.

- (e) (2 marks) If `s1 == "caption"` and `s2 == "cap"` how many characters need to be deleted from `s1` to transform it into `s2`? How many characters need to be inserted?
- (f) (2 marks) If `s2` is a subsequence of `s1` write down a formula, in terms of `s1.length()` and `s2.length()` the minimum edit distance between `s1` and `s2`.
- (g) (2 marks) If `s1 == "carrot"` and `s2 == "article"` how many characters need to be deleted from `s1` to transform it into `s2`?; how many characters need to be inserted?
- (h) (4 marks) Write down, in terms of `s1.length()`, `s2.length()` and `lcss(0,0)`, the minimum edit distance between `s1` and `s2`.
- (i) (7 marks) Write an efficient program to compute the minimum edit distance between arbitrary strings `s1` and `s2`. Your program may call functions `lcss`, `s1.length()` and `s2.length()`. **Your program should contain no more than two lines of C++ code.**

```
int minEdit(string s1, string s2);
// POST: returns the minimum edit distance between s1 and s2.
```

Write down the complexity of your program in big Oh notation, in terms of the lengths of the strings `s1` and `s2`. You should take into account the complexity of the function calls, but you may assume that the complexity of the string function `length` is $O(1)$. Explain your answer. (**Write no more than five lines.**)

3. (30 marks in total.)

This question concerns radix sort discussed in lectures, specialised to sort arrays of binary integers. (A binary integer is an integer expressed in binary notation.) We say that a binary integer is represented by d digits if its representation uses no more than d digits. For example 101 is a three-digit binary integer, but so is 10. (Recall that 10 as a three-digit integer is strictly speaking represented by 010.) We say that any d digit binary integer has d *attributes* (one for each digit). (Recall that we number the attributes from right-to-left, so that in 100, the right-most digit 0 is the 0'th attribute, the middle digit 0 is the 1'st attribute, and the left-most digit 1 is the 2'nd attribute.)

Let $A[0..n-1]$ be the array of d -digit binary integers to be sorted. The following program implements radix sort for this case.

```
void radixSort(int A[], int n, int d) {
// PRE: A is an array of d-digit binary integers; n is the length of A;
//POST: A is sorted.

    for (int j= 0; j < d; j++) { // Attribute loop: for each attribute j ...
        int G0[n]; int count0= 0; // digit is 0
        int G1[n]; int count1= 0; // digit is 1
        for (int i= 0; i < n; i++) { //Sorting phase: sort items according to attribute ...
            int k= digit(A[i], j);
            switch (k) { // .. depending on the digit, add to the matrix entry
                case 0: { G0[count0]= A[i]; count0++; break;}
                case 1: { G1[count1]= A[i]; count1++; break;}
            } // End of switch
        } // End of sorting phase
        // (*) Copying phase: copy items back into A from the least to the greatest digit...
        int y= 0;
        for(int k= 0; k < count0; k++) {
            A[y]= G0[k]; y++;
        }
        for(int k= 0; k < count1; k++) {
            A[y]= G1[k]; y++;
        } // (**) End of copying phase
    } // End of attribute loop
}

int digit(int a, int b);
// PRE: a is a (binary) integer, and b is at least 0
// POST: returns the b'th digit of a
```

(a) (8 marks) suppose that $A = \{101, 0, 11, 10, 1, 110, 11\}$ and d is 3. Trace the above algorithm for this array, **showing only** the status of the arrays G_0 , G_1 and A at points (*) and (**) (i.e. after the first and third inner loops).

(b) (2 marks) How many array look ups of A are required to sort this example?

- (c) (5 marks) Now write down a formula in terms of d and n which expresses the worst case complexity in big Oh notation of sorting an array of length n containing d -digit binary integers. You may assume that the most significant operation is lookups to the array A . Explain your formula. (**Write no more than three lines.**)
[Hint: How many times is the attribute loop executed? For each execution of the attribute loop, how many array lookups of A are there?]
- (d) (6 marks) The fastest known sorting algorithms have worst case complexity $O(n \log n)$, where n is the length of the array. By referring carefully to your answers above, explain whether `radixSort` is an example of such a fast algorithm for sorting arbitrary integers. Be careful to explain the role of the attribute parameter d .
- (e) (9 marks) The arrays `G0` and `G1` require the use of extra space. Using integer variables `g0`, and `temp` rewrite `radixSort` so that the extra arrays are not needed, and the final copying phase can be removed completely. Ensure that your code is carefully commented. Estimate the worst-case complexity of your final algorithm in Big-Oh notation.
[Hint: Consider using A directly, partitioning it with variable `g0`, and using `temp` as a temporary store for a value $A[i]$.]

End of Section A
Use a SEPARATE BOOK for Section B