Practice Questions for the Final

1. A undirected graph G has vertices a, b, c and edges {a,b}, {b,c}, {a,c}.

   a. Give the adjacency matrix M for G. (Assume the first row is for vertex a, second for vertex b, third row for vertex c and similarly for columns.) Also compute M^4 (Hint: $M^4 = (M^2)^2$.)

   b. How many paths of length 4 are there from vertex a to itself? _____

   c. How many path of length 4 from vertex b to vertex c? _____

2. Here are the vertices and edges of directed graph G: V = {a,b,c,d,e,f} E = {ab, ac, af, ca, bc, be, bf, cd, ce, de, df}. Weights: w(ab) = 7, w(ac) = 5, w(af) = 14, w(ca) = 2, w(bc) = 4, w(be) = 12, w(bf) = 10, w(cd) = 9, w(ce) = 7, w(de) = 3, w(df) = 2.

   a. Draw the Graph. This is a directed, weighted graph so you need to include arrows and weights.

   b. Apply Dijkstra's algorithm starting from vertex **b**. (If there is ever a tie for which edge to add next from the fringe list, choose the edge with the alphabetically smaller terminal vertex. When deciding whether replace one edge in the fringe list by another, only replace if the new edge has a strictly smaller number than the old one.)

STEP	1	2	3	4	5	6
verts. cur. in tree						
fringe list						
next edge to add						

   c. Draw the shortest path tree.

d. Consider the Graph H which is an undirected graph with the same edges as G (except they are now unordered pairs rather than ordered pairs). Also we will throw out edge ca (since there's already an edge ac in G). Draw the minimum spanning tree for H and list the edges in the order in which they were inserted.

3. Implement a function called dfsHelper to be called by the dfs function below. dfsHelper should traverse the graph in a depth first manner via recursion. After returning to dfs the parents array will have recorded a depth first traversal from the given starting vertex.

,

```
void dfs (vector<int> alists[], int size, int start) {
        int * parents = new int[size];
        for (int i = 0; i< size; i++) parents[i] = -1;
        parents[start] = start;
        dfsHelper(alists, size, start, parents);
        for (int i = 0; i < size; i++)
                cout << parents[i] << " ";
        cout << endl;
        delete [] parents;
}
```

4. Recall that an array maps to a complete binary tree using the correspondence: *2i+1, 2i+2* → *left child, right child*. Using this mapping, implement the function `bool isMaxHeap(int nums[], int root, int size)` that returns `true` if the subtree of `nums` with root *root* has the maxheap property and `false` otherwise. For example, if `nums` is `{ 2, 7, 5, 7, 2, 4, 5, 3 }` then `isMaxHeap(nums, root, 8)` would return `true` for values of *root* between 1 and 7 but false if *root* is 0.

   You may assume that the initial call to `isMaxHeap` has `0 ≤ root < size`.

5. We wish to hash five-by-five tic-tac-toe boards. Each board is a five-by-five array of char in which array elements are guaranteed to be either 'X', 'O' or '–'. Here is a proposed hash function:

```
int hash(char board[][5], int tableSize) {

    int h = 0;

    for (int i = 0; i < 5; i++)

        for (int j = 0; j < 5; j++)

            if (board[i][j] == 'O') h+=1;

            else if (board[i][j] == 'X') h+=2;

    return h % tableSize;

}
```

Why will this hash function probably generate a lot of collisions even if `tableSiz` is big? How can it be fixed?

6. In quadratic probing (when clustering is avoided) the expected number of probes for an unsuccessful search is $1/(1-\alpha)$ where α is the load factor (number of items stored divided by table size). In separate chaining, the expected number of probes is just α. Assuming runtime is proportional to number of probes…

   a. using quadratic probing, how much longer will unsuccessful searches take, on average, when α goes from 0.5 to 0.9.

   b. using separate chaining, how much longer will unsuccessful searches take, on average, when α goes from 0.5 to 0.9.

7. Klotski.

In the puzzle "Klotski", up to `N^2` blocks occupy an N by N tray. Each block has its upper left corner at a particular position in the tray (e.g. position (0, 2) means left side, down two units from the top). Blocks can be four different sizes since width, height $\in \{1,2\}$. The goa is to slide blocks around until a special block reaches the exit position. (A related game is the 15 puzzle .) To solve the puzzle, we try all possible moves and backtrack when a previous position is reached. To avoid loops we must store all previously reached positions, so a hash table makes sense here.

```
// Block type
struct Block {
      int width, height;
      bool special; // true if the block is special, false otherwise
};

// a global constant
const int N = 8;  // N might be another number, not necessarily 8

// the positions of Blocks in the tray
Block * positions[N][N];  A pointer is null if no block occupies that position.
```

Implement the hash function below. Distinct configurations should map to distinct indexes, assuming `tableSize` is big enough. (Perfect hash function.)

```
unsigned int hash(Block * positions[][N],  unsigned int tableSize)
```