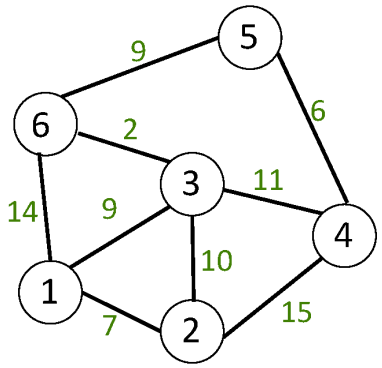
Notes by Karina Macancela
April 20, 2016

Homework 8:

1. Show how Kruskal's Algorithm would build a Minimum Spanning Tree for the graph below. Apply an appropriate sort for the edge-weights, then perform the six MAKE-SET operations, finally do the necessary number of FIND-UNION operations. Compare the MST obtained with the one obtained in last week's homework using Prim's Algorithm.

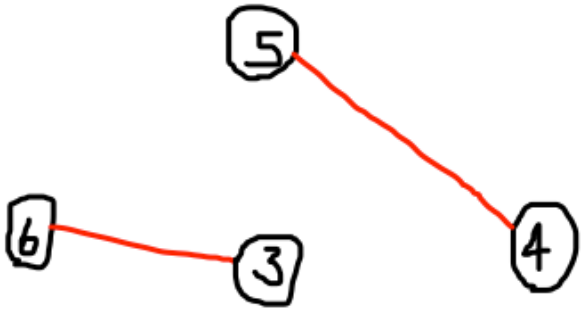


There are 9 edges made up of 6 vertices. We can make the following Edge x Weight table:

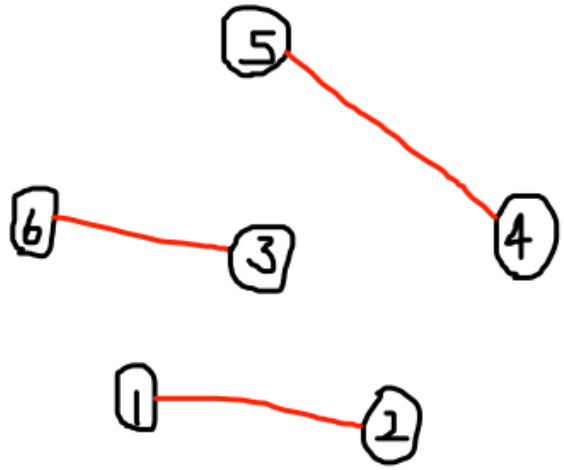
Edge	Weight
3 to 6	2
4 to 5	6
1 to 2	7
1 to 3	9
5 to 6	9
2 to 3	10
3 to 4	11
1 to 6	14
2 to 4	15

First, we select the minimum weight to be 2 (edge 3 to 6):

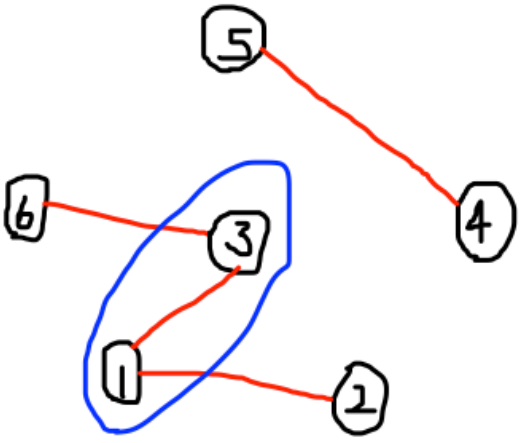
Then, we select the minimum weight to be 6 (edge 4 to 5):



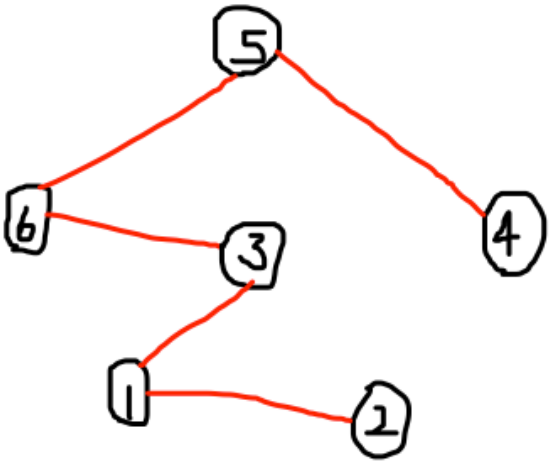
Then, we select the minimum weight to be 7 (edge 1 to 2):



Then, we select the minimum weight to be 9 (edge 1 to 3):



Then, we select the minimum weight to be 9 (edge 5 to 6):



We stop here because we have 6 vertices so our Minimum Spanning Tree can only have 5 edges.

2. Consider a rectangular m x n grid which whose cells each contains either a 0 or 1. Design an O(m*n) algorithm that will find the largest square k x k sub-grid whose cells contain only 1s. (Hint: use an auxiliary array that keeps track of the largest all-1s square whose upper-left corner begins at that position.)

```
int i, j;
int S[ k ][ m ];
for (i = 0; i < k; i++){
        S[ i ][ 0 ] = A[ i ][ 0 ];
}
for (j = 0; j < m; j++){
        S[ 0 ][ j ] = A[ 0 ][ j ];
}
for (i = 1; i < k; i++){
        for (j = 1; j < m; j++){
                if (A[ i ][ j ] = 1)
                        S[ i ][ j ] = min(S[ i ][ j-1 ], S[ i-1][ j ], S[ i-1 ][ j-1 ]) + 1;
                else
                        S[i, j] = 0;
        }
}
```

3. Consider a set of n items having respective weights (w_1, w_2, …, w_n) and values (v_1, v_2, …, v_n). Design an algorithm that finds the subset of these items that has maximal value, but without

exceeding some integer total weight constraint W. The algorithm should run in O(n*W), where again, W is the integer upper bound on the total weight (capacity) of the container. Note that in this "0-1" version of the problem, you can either include an item or exclude it; you cannot include multiple copies of a given item. (Hint: as you gradually allow yourself to use one more item, that will also reduce the remaining allowed weight remaining in the container.)

"0-1" knapsack

```
for w = 0 to W
        do V[ 0 ][ W ] <- 0
for i = 0 to n
        do V[ i ][ 0 ] <- 0
        for w = 0 to W
                do if w_i <= w && V_i + V[ i-1 ][ w-w_i ] > V[ i-1 ][ w ] then
                        V[ i ][ w ] <- V_i + V[ i-1 ][ w-w_i ]
                else
                        V[ i ][ w ] <- V[ i-1][ w ]
```

4. A polygon is a geometric figure in the plane with at least three straight sides and angles between them. (With 3 sides, it is a triangle; with 4 sides, it is a quadrilateral; with 5 sides, it is a pentagon; with n sides, it is an n-gon.) We would like to "triangulate" the polygon, i.e. draw lines between vertices of the polygon so that the entire polygon is now divided up into triangles. Each of the triangles has a weight associated with it: if a triangle has vertices A, B, C, then the weight of triangle ABC is defined as d[A, B] + d[B, C] + d[C, A], where d is the geometric distance between the two points. Design an algorithm that finds the optimal triangulation, i.e. the division of the polygon into triangles so that the sum of the weights of all the triangles is the minimum possible.

We let the minimum cost of triangulation of vertices to go from I to j and call it min_cost[i][j] and let k vary from i+1 to j-1:

```
if j <= i+2 then
        min_cost[ i ][ j ] =0
else
        min_cost[ i ][ j ] = min (min_cost[ i ][ k ] + min_cost[ k ][ j ] + cost[ i ][ k ][ j ])
```

The cost of a triangle formed by the edges [i][j], [j][k], and [k][j]:

```
cost[ i ][ j ][ k ] = dist[ i ][ j ] + dit[ j ][ k ] +dist[ k ][ j ]
```

Review for Exam 2:

1. Hash function
    - -Pros and cons
    - -2 solutions: probing and chaining
    - -Time complexity: is constant time
2. Floyd's Algorithm

```
for i = 1 to n
    for j = 1 to n
        d[i , j] = c[i , j]
        p[i , j] = i
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            if (d[i , k] + d[k , j]) < d[i , j] then
                d[i , j] = d[i , k] + d[k , j]
                p[i , j] = p[k , j]
```

3. Huffman coding
4. Hashing
5. Trees
    - -Binary Search
    - -etc
6. Heaps
    - -Fibonacci
    - -Binary
    - -No code, just features and know how it affects performance
7. SSSP
    - -Dijkstra's: time complexity?...depends…O(v^2) adjacency list is better than adjacency matrix because focuses on neighbors that do exist
    - -Belman Ford: will work for negative edges; time complexity? O(V*E)
    - -Neither will work for what?
8. MST Property
    - -Said: "If I have 2 sets and ___ use min cost ___ if the 2 algorithms."
    - -Borrurka (did not discuss in class)
    - -Prim
    - -Kruskal (uses the disjoint set)
9. C-Find
10. W-Union
11. APSP – Floyd
12. Transitive Closure – Warshall
    - -Said: "There exist a path between 2 vertice…Not concerned with the cost."
13. Dynamic Programming

-Fibonacci numbers
-Combinations e.g. Pascal's formula: $C(n , 1) = C(n-1 , r-1) + C(n-1 , r)$
-Making change
-Floyd
-Prim
-Dijikstra
-Max subsequence/sequence variations
-Should be able to solve problems with them.

In Today's Lecture:

We discussed Divide and Conquer (https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms) and Big Integer Multiplication (http://people.mpi-inf.mpg.de/~mehlhorn/ftp/chapter2A-en.pdf) .

First we must under that if we are working with binary numbers, then we can say the following, where x_L is the left hand side if we split x into two pieces and x_R is the right hand side:

$$x = x\_L + x\_R$$
$$x = 2^{(n/2)}\, x\_L + x\_R$$

If we are given two binary number, each of length n, then to multiply them we can use the following algorithm:

$$xy = (2^{(n/2)}\, x\_L + x\_R)(2^{(n/2)}\, y\_L + y\_R)$$
$$xy = (2^{(n/2)}\, x\_L)(2^{(n/2)}\, y\_L) + 2^{(n/2)}\, x\_L\, y\_R + 2^{(n/2)}\, x\_R\, y\_L + x\_R\, y\_R$$
$$xy = 2^{(n/2)}\, x\_L\, y\_L + 2^{(n/2)}[x\_L\, y\_R + x\_R\, y\_L] + x\_R\, y\_R$$

The last line is what we would use. Hence the method we can conclude with is the following:

1. Make recursive calls to multiply these 4 pairs of (n/2)-bit numbers (i.e. x_L y_L, x_L y_R, x_R y_L, and x_R y_R).
2. Evaluate the preceding expressions in O(n) time.
3. Over all running time on n-bit inputs is $T(n) = 4T(n/2) + O(n)$