Mid Term Exam for CPSC 5210: Algorithm

1. (8 points) Use **dynamic programming** to solve the **single-destination shortest path** problem for a directed graph is to find the shortest path from every vertex to a specified destination vertex **v**. Give an efficient algorithm (pseudo code) to solve the **single-destination** shortest paths problem and discuss the time complexity.

Solution:

If we **reverse** the directed edges in the graph, the problem reduces to **Single Source Shortest Path problem.** This can be solved using Bellman Ford's algorithm as it is a Dynamic Programming .

Algorithm /Pseudo code:

```
Step1: Procedure BellmanFord(g, src)
Step2: For (i =0;i<V;++i) // for each vertex V belongs to g do
Step3:     dist[i] = infinite
Step4:     dist[src] = 0
Step5: end for
Step6: for each vertex V belongs to g do
Step7:      for each edge U, V belongs to g do
Step8:           u = g.edge[ j ] .src and v = g. edge [ j ] . dist and weight = g.edge [ j ] .weight
Step9:            tempdist1[ V ] = dist [ U ] – edgeWeight ( U , V)
Step9:             if tempdist1 > dist [ V ] then
Step10:                  dist [ V ] = dist [ V ]
Step 11 :                dist [ src ] = U
```

```
Step12 :              end if
Step 13:      end for
Step 14: end for
Step15: for each vertex V belongs to g do
Step16:        return dist[]
Step17: end for
Step18: end procedure
```

tempdist1[V] = dist [U] – edgeWeight (U , V)and if tempdist1 > dist [V] will go through the single destination shortest path.

Time Complexity:
The run- time complexity for the Bellman-Ford is the O(VE), as it travels all the vertices and edges.

2. (16 points) Firstly, compute the time-complexity of the following algorithm (9 points). Secondly, re-implement this algorithm using the most efficient method that you can imagine and investigate its time complexity (7 points).

```
public static long foo(int n) {
        if (n<=1) return 0;
        else if (n==2) return 1;
        else return foo(n-1) + 3*foo(n-2) + 2*foo(n-3);
};
```

Solution:

Time complexity for the above code:

F(n) = 0 if $n \le 1$

1 if n =2

f(n-1)+ 3f(n-2) + 2f(n-3) if n > 2

So,

T(n) = T(n-1) + 3T(n-2) + 2T(n-3) + 2

$\le$ T(n-1) + 3T(n-1) + 2T(n-1) + 2

So, we will take it as

T(n) < 8T(n-1) + 2 (as 3 and 2 are constants)

2^{0} [T(n) = 8 T(n-1) +2]----------------(0)

2^{1*3}[T(n-1) = 8 T(n-2) + 2]------------------(1)

2^{2*3}[T(n-2) = 8 T(n-3) + 2]------------------(2)

.

.

.

.

$2^{3*(n-3)}$[T(n-(n-3)) = 8 T (2) + 2] -------(n-3)

Without loss of generality ,

(1) + (1) + (2) +...........+(n-3)

$$T(n) = 2^{0} * 2 + 2^{1*3} * 2 + 2^{2*3} * 2 + ... + 2^{3*(n-3)} * 2$$

$$= 2 \sum_{k=0}^{n-1} 2^{3i}$$

$$= O(2^{n})$$

The efficient method to re-implement this is using dynamic programming,

```
Public static long foo(int n) {

    int fo[] = new int[n+1];
    f[0] = 0;
    f[1] = 0;
    f[2] = 1;
    for (int k = 3; k <= n; k++)
    {
    fo[k] = fo[k-1] + 3*fo[k -2 ]+2 * fo[k-3];
    }
    return f[n];

}
```

Time complexity for the above code:

T(n) = O(n)

3. (24 points) You are asked to modify the LCS algorithm to identify the **heaviest common-subsequences (HCS)** using a weighted scoring scheme, denoted as matrix M.

	A	B	C	D
A	1	0.4	0	0
B	0.4	2	0	0
C	0	0	1	0.7
D	0	0	0.7	1

Table 1: Scoring scheme described by matrix M

Based on this score scheme, A-A, C-C, and D-D match has a score of 1, B-B match has a score of 2, A-B match has a score of 0.4, C-D match has a score of 0.7.

Given that two sequences:

X = **BACBADCDD** and Y = **BCDABCBDD**,

3(A): (12 points): First, illustrate the application of dynamic programming on heaviest common sub-sequence (HCS) using the example of X and Y sequences given above, by following the example of Figure 15.8 in the CLRS textbook.

Solution:

	Y	B	C	D	A	B	C	B	D	D
X	0	0	0	0	0	0	0	0	0	0
B	0	2	2	2	0.4	2	2	2	2	2
A	0	0.4	2	2	3	0.8	2	2.4	2.4	2.4
C	0	0.4	1.4	2.7	3	3	1.8	2.4	3.1	3.1
B	0	2	2	2.7	3.1	5	5	3.8	3.8	3.8
A	0	0.4	2	2.7	3.7	3.5	5	5.4	5.4	5.4
D	0	0.4	1.1	3	3.7	3.7	4.2	5.4	6.4	6.4
C	0	0.4	1.4	1.8	3.7	3.7	4.7	5.4	6.1	7.1
D	0	0.4	1.1	2.4	3.7	3.7	4.4	5.4	6.4	7.1
D	0	0.4	1.1	2.1	3.7	3.7	4.4	5.4	6.4	7.1
		B			**A**	**B**		**B**	**D**	**D**

Heaviest common sub-sequence(HCS) : **BABBDD**

3(B): (12 points) Design a HCS algorithm by modifying the LCS-LENGTH(X, Y) in Chapter 15 of the CLRS textbook using a weighted scoring matrix M as shown above.

Solution:

Algorithm:

```
Step1: Procedure HCS(X,Y,M)
Step2: m = x. length
Step3: n = y. length
Step4:m1 = mx. length
Step5:m2 = my. length
Step6:let b[1...m, 1 ...n] and c[0...m , 0 ..n]
Step7: for i = 1 to m
Step8:     c[i,0] = 0
Step9: end for
Step10: for j = 0 to m
Step11:     c[ 0, j ] = 0
Step12:end for
Step13:for i = 1 to m do
Step14:     for j = 1 to m
Step15:         for p = 0 to m1
Step16:             for q = 0 to m2
Step17:                 if M[p][q] > 0
Step18:                         k = M[ p ][ q ]
Step19:                         c[i , j] = c[ i-1,j-1] +k
Step20:                         b[i, j] = right arrow
Step21:                 else if c[ i-1 , j ] ≥ c[ i, j -1 ]
Step22:                         c[i, j] = c[i-1, j]
Step23:                         b[i ,j] = upward arrow
Step24:                 else c[ i, j] = c[i, j-1]
Step25:                         b[ i, j] = left arrow
Step26:                 end if
Step27:             end for
Step28:         end for
Step29:     end for
Step30: end for
Step31: return c and b
Step32:end procedure
```

4. (8 points) Design a BFS-based algorithm (pseudo code) that computes the length of the first detected circle. Investigate the related time complexity.

Solution:

```
Step1: procedure bool isconnected(vector<int> a, int s, int v, vector<bool> &visited)
Step2: create a queue as q to store adjacent element for every vertex
Step3: vector<int> parent (v , -1) as the parent vertex of every node is set to -1 first
Step4:set visited [ s ] as true and q. push(s)
Step5: while (!q. empty()) do
Step6:       u = q. front(), count = 0 and pop()//
Step7:       for ( auto v : adj [ u ] )// we are picking a vertex and going through the vertex that
i                       is adjacent to it
Step8:           if( ! visited[v ])
Step9:              do count++
Step10:             set visited[v] = true // if adjacent isn't visited then marking it as visited
Step11:            q. push(v) and parent[v] = u // parent is marked because it should not be
included in the cycle
Step12:          else if parent[u] ! v // if it's not the parent and has already been visited , then
it's a cycle
Step13:             return true;//returns the length of 1st detected cycle which is "count".
Step14:     end for
Step15: end while
Step16: return false
Step: end procedure
```

The time complexity of BFS is O(V+E)

5. (8 points) Implement a database for a new telephone company. The database will contain the names and phone number of all its clients. For the operation, the company wants the average running time to be as small as possible: (1) insert a new entry, (2) search for a person's phone number, and (3) remove an entry from the database. What data structure will you use, and what are the main design issues that you need to consider?

Solution:

The data structure that will be used is Hashing.(Hash table)

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
A good hash function should have following properties
1) Efficiently computable.
2) Should uniformly distribute the keys .

Hash table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

The main design issues that need to be considered:

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

Collision handing techniques:

1. **separate chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple but requires additional memory outside the table.
2. **Open addressing:** In Open Addressing, all elements are stored in the hash table itself. So, at any point, size of the table must be greater than or equal to the total number of keys . It can be done in following ways:
    1. linear probing
    2. quadratic probing
    3. double hashing.

6. (15 points) 0-1 Knap-sack problem. Illustrate the application of dynamic programming on the following Knapsack problem with the maximum weight of 8 pounds. **Please note that chargers and their served devices should always be taken together.**

Item	Value (dollars)	Weight (pounds)
Cell phone	100	1
Charger for Cell phone	30	1
Laptop	300	3
Charger for Laptop	70	1
Tent	250	4

Medicine	80	1
Instant Food	80	2
Water bottle	15	1
Umbrella	25	1
Flashlight	45	1

Solution:

After combining the chargers and their served devices the table looks like:

Item	Value(dollars)	Weight(pounds)
Cell phone + charger for cell phone	130	2
Laptop + charger for laptop	370	4
Tent	250	4
Medicine	80	1
Instant food	80	2
Water bottle	15	1
Umbrella	25	1
Flash light	45	1

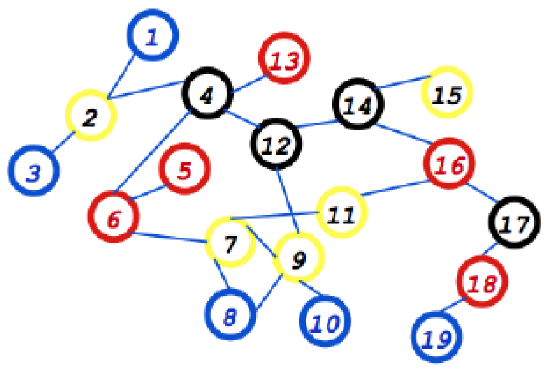
Knap Sack Problem:

Item	Value	Weight	0	1	2	3	4	5	6	7	8
Cell phone + charger for phone	130	2	0	0	130	130	130	130	130	130	130
Laptop + charger for laptop	370	4	0	0	130	130	370	370	500	500	500
Tent	250	4	0	0	130	130	370	370	500	500	620
Medicine	80	1	0	80	130	210	370	450	500	580	620
Instant Food	80	2	0	80	130	210	370	450	580	580	620
Water bottle	15	1	0	80	130	210	370	450	580	580	620
Umbrella	25	1	0	80	130	210	370	450	580	580	620
Flash light	45	1	0	80	130	210	370	450	580	580	**625**

Hence , the maximum value is 625 which comes from choosing Flashlight , Medicine ,

Cell phone + charger and Laptop + charger.

7. (15 points) Develop algorithm for graphs with colored nodes. An example of undirected graph with colored nodes is given below. In this kind of colored-graph ***G(V;C;E)***, we assume that there are ***V*** number of nodes, ***C*** number of colors in the nodes, and ***E*** number of links.



(a) (10 points) develop a single-source shortest-path algorithm for a node that can travel to the closest nodes with different colors. When calculating traveling distances, path between different colors are consider as 1, and paths between the same-colored nodes is considered as 0 (zero).

Modify the textbook algorithms of DIJKSTRA(G,w,s) on page 658 and RELAX (u,v,w) on page 649 in Chapter 24 to ***Color-DIJKSTRA(G, w, s)*** and ***Color-RELAX (u,v,w)*** .

Solution:

Algorithm : *Color-DIJKSTRA(G, w, s)*

```
Step1: procedure Color-DIJKSTRAG(G,w,s)
Step2: Initialize-Single-Source(G,s)
Step3: S = Ø; and initialize color as true and distance d =0
Step4: W = G.V
Step5: while Q != Ø
Step6:     u = Extract-Min(Q)
Step7:     S = S U {u}
Step8: end while
Step9: for each vertex v  ∈ G.Adj[u] do
Step10:      color-Relax(u,v,w)
Step11: end for
Step12: end procedure
```

Algorithm : ***Color-RELAX (u,v,w)***

```
Step1: procedure Color-Relax(u,v,w)
Step2: if v.d > u.d + w(u,v) && v.color != u.color then
Step3:       v.d = u.d + w(u,v)
Step4:       v.π = u
Step5: end if
Step6: end procedure
```

(b) (5 points) Discuss the worst-case running time for ***Color-DIJKSTRA(G, w, s)*** with ***Color-RELAX(u,v,w).***

Solution:

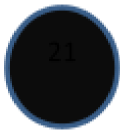
The worst-case running time for Color-DIJKSTRA(G, w, s) *is* O(V^2) as all the vertices should be visited. For both Color-DIJKSTRA(G, w, s) with Color-RELAX(u,v,w) the worst-case running time would be O($2V^2$).

8. (6 points) (Red-Black Tree) Show the red-black tree that results after each of the integer keys 21, 32, 64, 75, and 15 are inserted, in that order, into an initially empty red-black tree. Clearly show the tree that results after each insertion (indicating the color of each node) and make clear any rotations that must be performed.

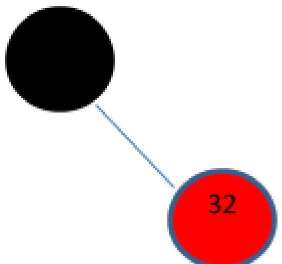
Solution:

Top-down approach:

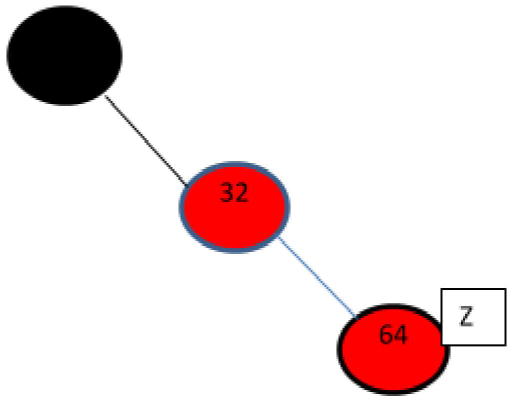
Step1: Insert 21



Step2: insert 32

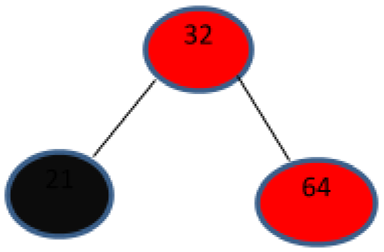


Step3: Insert 64

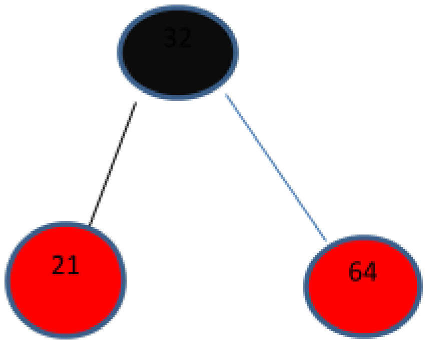


Applying case 3 (Z uncle is black then rotate grandparent and recolor Z’s uncle is 21’s left child which is NIL and is in black).

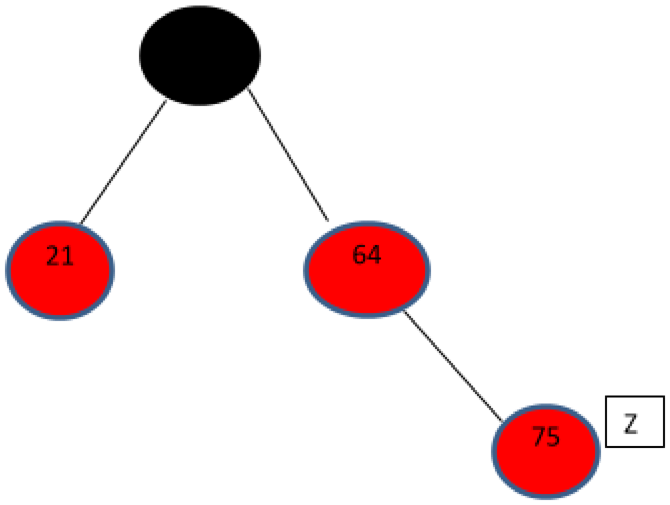
After Rotation:



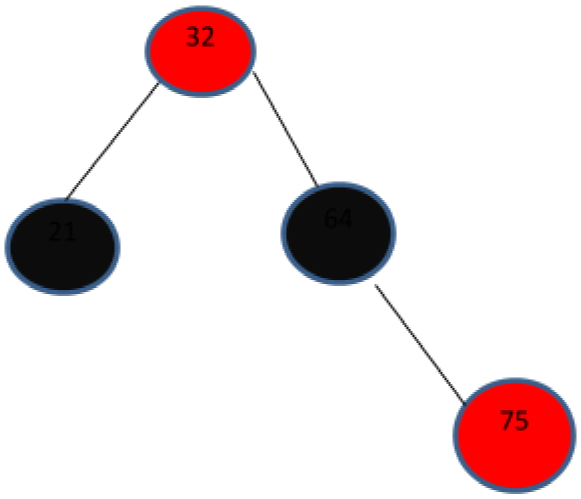
Recolor the nodes



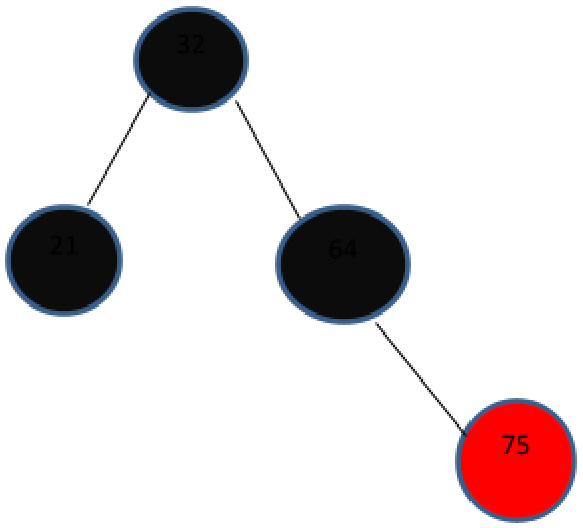
Step4: Insert 75



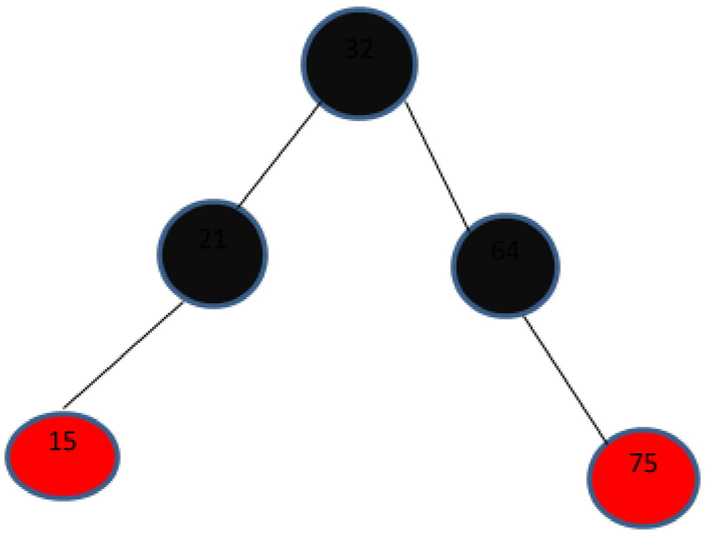
Apply case 1(Z.uncle is red then recolor (Here Z’s uncle is 21)



Apply case 0(root should be black)



Step 5: Insert 15


32
21
64
15
75