**Question 1 Give an algorithm that sorts (into non-decreasing order) an input array of n integers in the range 0 to n3 − 1. Your algorithm must run in O(n) time.**

We can use a modification of Radix sort. Instead of sorting in radix of 10, we will sort in radix n, such that the sorting will be done in 3 passes. The following c code will perform the operation in 5 scans of the array, therefore in approx. 5n time, or O(n). The code was compiles using gcc.

#include <stdio.h>

// Sorting function

void radixSort(int n, int A[n]){

int i, pass, digit, PASSED, mod = 1, devidor = 1;

int countArray[n];

int tempArray[n];

// Run the sort 3 times

for(pass = 1; pass <= 3; pass++){

mod \*= n;

// Format the couting array (takes n operations)

for(i = 0; i< n; i++)

countArray[i] = 0;

// Count who gets to go where (takes n operations)

for(i = 0; i<n; i++)

countArray[A[i]%mod/devidor]++;

// Make it cumulative count arrays (takes n ops)

for(i = 0; i< n; i++)

countArray[i+1] += countArray[i];

//Perform radix sort with the information provided in the cointing array (takes n ops)

for(i = n-1; i >= 0; i--)

tempArray[--countArray[A[i]%mod/devidor]] = A[i];

// Copy the temp array into the original array (takes n ops)

for(i = 0; i< n; i++)

A[i] = tempArray[i];

devidor \*= n;

}

// Print the sorted result an run coherency test

PASSED = 1;

printf("\nSorted array: ");

for(i = 0; i< n; i++){

printf("%d ", A[i]);

if(i != n-1)

if(A[i] > A[i+1])

PASSED = 0;

}

printf("\n");

if(PASSED)

printf("PASSED test with n = %d\n", n);

else

printf("FAILED test with n = %d\n", n);

}

// testbench

int main(void){

int n, i, range;

int \*array = malloc(sizeof(int));

// Testbench => feel free to increment n

for(n = 4; n<=20; n++){

range = n\*n\*n;

// Allocate memory

free(array);

\*array = malloc(n\*sizeof(int));

// Populate array

printf("Unsorted Array: ");

for(i = 0; i<n; i++){

array[i] = (rand()%range);

printf("%d, ", array[i]);

}

// Run algorithm

radixSort(n, array);

printf("\n");

}

return 0;

}

**Question 2 For this question, an arithmetic expression (or just expression) is built from integers and variables x1, x2, . . . , using the operations +,−,×,÷ as follows:**

* **any number is an expression,**
* **any variable is an expression,**
* **if A and B are expressions, then so are (A − B) and (A ÷ B),**
* **if A1, A2, . . ., Ak are expression, then so are (A1 + A2 + . . . + An) and (A1 × A2 × . . . × An)**

**For example, ((x1 + 5 + (x2 × 3 × x6) + (x2 ÷ x1)) − 4) is an expression.**

**(a) Give a data structure for representing arithmetic expressions as trees of unbounded branching.**

**Clearly explain the fields you are using.**

To use a tree of unbounded branching, each node X will have the following components:

* A pointer o the parent of X, p[x]
* A pointer to the left child of x, pointing to the leftmost child of node, lc[x]. If the node has no children, then lc[x] = NULL
* A pointer to the right sibling of x, pointing to the sibling of x immediately to the right of it, rs[x]. If x is the rightmost child, then rs[x] = NULL
* 2 Integer fields:
  + The first one called ‘value’: filled with an integer value, if this node is to be an integer, its value will be recorded here. If the node is to be a variable, value = NULL. If the node is not a leaf node (indicated by lc[x] = NULL), then the node is to represent an arithmetic expression. We will adopt the convention 1 = ‘+’, 2 = ‘-‘, 3 = ‘\*’ and 4 = ‘/’. By using this convention to record arithmetic expressions, we have a lot of freedom to expand, as new operations like square root ant ‘^’ can easily be coded here
  + The second one called ‘index’: integer containing the index of the associated variable in the X array. For example, if this node is to be represented by X[3], then this field would contain var = 3.

The tree will be constructed using a standard expression tree, but with each node with an unlimited number of children. Because of the nature of an expression tree, each leaf node will represent either an integer or a variable. Every non-leaf node will be assumed to be representative of an arithmetic expression, so the algorithm will only take into consideration the ‘value’ field, with the convention proposed earlier.

Note: non-leaf nodes, since they represent an arithmetic operation, MUST have at least 2 children in a coherent representation. Otherwise, we would have a situation like:

(5+2)\*(4+’MISSING’)

**(b) Give an algorithm that on input (A,X), where A is the root of the tree representing an expression which we also call A and X is the array of the values for variables, outputs the value of expression A when the variables are set according to X (i.e., x1 = X[1], x2 = X[2], etc.).**

// Defines as per our chosen convention

#define PLUS 1

#define MINUS 2

#define MULTIPLICATION 3

#DEFINE DIVISION 4

// Data structure definition

struct node{

node parent; // Pointer to the parent node

node leftChild; // Pointer to the leftChild

node rightSibling; // Pointer to the right Sibbling

int value; // Value of integer if leaf, operation of non-leaf

int index; // Index for array of varialbes X[index]

};

// Assume there already exists a tree built following our data structure.

int evaluate(A, x){

// Following variable will contain the result

int result = 0;

// We will use a temp node to look and evaluate children

node tempNode;

// for simplicity, this variable will be used to evaluate non-leaf nodes

// Not absolutely nescessaity, but makes the code more readable

int valueChild;

// We know that a non leaf node must have at least 2 children

// Start evaluating from the leftmost child

tempNode = A.leftChild;

// First, extract the value of the tempNode or compute it recursively

// Storing the result in 'result' variable

if(tempNode.leftChild == NULL){ // If our node is a leaf

if(tempNode.value != 0) // If the leaf represents an integer

result = tempNode.value; // Extract the integer value

else if(tempNode.value == 0) // If leaf node represents a variable

result = X[tempNode.index]; // Evaluate variable using index

}

else if(tempNode.leftChild != NULL) // If the node is not a leaf

result = evaluate(tempNode, X); // Call recursively to evaluate the node

// At this point, result contains the value of the first leftchild of the root,

// Or the whole left subtree if it exists

// Evaluate entire lower level, knowing that any non leaf node will have at least 2 child, so this

// while loop will get executed at least once

while(tempNode.rightSibling != NULL){

// Jump to the next sibbling on the right

tempNode = tempNode.rightSibling;

// Extract the value of the tempNode or compute it recursively

if(tempNode.leftChild == NULL){ // If our node is a leaf

if(tempNode.value != 0) // If the leaf represents an integer

valueChild = tempNode.value; // Extract the integer value

else if(tempNode.value == 0) // If leaf node represents a variable

valueChild = X[tempNode.index]; // Extract value from variable using index

}

else if(tempNode.leftChild != NULL) // If the node is not a leaf

valueChild = evaluate(tempNode, X); // Call recursively to evaluate the node

// Now result is used as an accumulator, accumulating valuChild as the while loop moves //forward. Since the function is ALWAYS called on a root node, the node 'A' always refers to //an operation

switch A.value{

case PLUS:

result += valueChild;

break;

case MINUS:

result -= valueChild;

break;

case MULTIPLICATION:

result \*= valueChild;

break;

case DIVISION

result /= valueChild;

break;

default:

printf("Detected invalid Sign!\n");

break;

}

}

return result;

}

**(c) Give an algorithm that given the root of the tree representing an expression prints out the expression.**

// Defines as per our chosen convention

#define PLUS 1

#define MINUS 2

#define MULTIPLICATION 3

#DEFINE DIVISION 4

// Data structure definition

struct node{

node parent; // Pointer to the parent node

node leftChild; // Pointer to the leftChild

node rightSibling; // Pointer to the right Sibbling

int value; // Value of integer if leaf, operation of non-leaf

int index; // Index for array of varialbes X[index]

};

// Assume there already exists a tree built following our data structure.

int printExpression(A){

// We will use a temp node to look and evaluate children

node tempNode;

// Following variable to record sign of the root; not necessary, but makes code easier to follow

int rootSign = A.value;

// We know that a non leaf node must have at least 2 children

// Start evaluating from the leftmost child

tempNode = A.leftChild;

// Print the lower level using a do/while loop such that the rightmost child

// gets printed before breaking the loop condition

do{

// First, extract & print the value of the tempNode or expression

if(tempNode.leftChild == NULL){ // If our node is a leaf

if(tempNode.value != 0) // If the leaf represents an integer

printf("%d", tempNode.value); // Print the integer value

else if(tempNode.value == 0) // If leaf node represents a variable

printf("X[%d]", tempNode.index);// Print variable and index

}

else if(tempNode.leftChild != NULL) { // If the node is not a leaf

printf("("); // Open bracket

printExpression(tempNode); // Call recursively to print sub-expression

printf(")"); // Close bracket

}

// Next print arithmetic expression if there is another right sibbling

if(tempNode.rightSibling != NULL){

switch rootSign{

case PLUS:

printf(" + ");

break;

case MINUS:

printf(" - ");

break;

case MULTIPLICATION:

printf(" \* ");

break;

case DIVISION

printf(" / ");

break;

default:

printf("Detected invalid Sign!\n");

break;

}

}

} while(tempNode.rightSibling != NULL)

}

**Question 3: (a) Clearly describe the data structure you are using.**

I would use a graph ‘G’, where every vertex represents a wrestler and every edge represents a rivalry. Since there are n wrestlers and r rivalries, G will contain n vertices and r edges.

The vertices data structure would contain the following elements

struct node{

node prev; //pointer to the previous element

int d; // distance metric for the start of a search algorithm, + 1 at every hop

bool VISITED; // a Boolean flag to indicate if the node has been visited

string orientation; // a string to label the wrestler heel or babyface

node adj; // Adjacency linked list.

}

**(b) Give the algorithm. (Your algorithm should consist of 3 parts: one for parsing the input, one for perform some graph search, one for print the output).**

**Summary**

Construct the graph by creating a node for all the wrestlers and populating the adj list with the rivalries. Once the graph is constructed, if the graph is connected, pick an arbitrary vertex ‘s’ and run a BFS on ‘s’. If the graph is not connected, run BFS on all the components.

After the BFS, iterate over every edge (u, v). If d[u] and d[v] are either both even or both odd, then the designation is not possible, print(“Impossible”). Otherwise, define all u=babyface if d[u] = even, and v = heel if d[v] = odd. If all the edges pass the test, then we know that it is a good designation because every rivalry is between a good guy and a bad guy.

The input to the algorithm is an array W of distinct names (of the wrestlers), and an array R of distinct pairs of rivalries.

// The BFS code taken from the course book, and because of its triviality will not be commented

BFS(s, G){

for each vertex u in V(G){

u.VISITED = FALSE;

u.d = 0;

u.prev = NIL

}

Q <= {s}

while Q != empty do

u <= head[Q]

for each v in Adj[u] do

if(!v.VISITED) do

v.VISITED = true;

v.d = u.d + 1;

v.prev = u;

Q.enqueue(v);

}

// The following code will perform the required task

designation(W, R){

// Define global variable to record possibility of match

bool POSSIBLE;

// part 1: Gather input and build data structure. First scan all the names of the wrestlers

for all elements w in W do{

new node guy in graph G; // Create a node per wrestler

w.prev = NULL;

w.d = INF;

w.VISITED = false;

string name = 'name string in w';

string orientation = NULL;

for all element r in R do{

if w is present in r

add rival to w.adj;

}

}

// Part 2: Graph search

for all vertices ‘guy’ in graph G, do{

BFS(guy, G) // Run BFS starting from all nodes

// After BFS, set a designation to all nodes as per convention

for all nodes ‘guy’ in G do{

if(guy.d%2) // If distance is odd

guy.orientation = "Heel";

else // Otherwise, if distance is even

guy.orientation = "babyface";

}

// Assume a coherent match

POSSIBLE = true;

// Check to ensure that the matches are indeed coherent

// Simultaneously, reset the visited flag to avoid double printing at the next step

for all nodes ‘guy’ in graph G do{

guy.VISITEd = false;

for all nodes in adjacency list of ‘guy’, do

if(guy.orientation == guy.adj.orientation)

POSSIBLE = false;

}

// Finally, if there is such a coherent possibility, print it

if(POSSIBLE)

for all nodes guy in graph G except root node for BFS

if (!guy.VISITED) do{

print("%s (%s) Vs %s (%s)", guy.name, guy.orientation,

guy.prev.name, guy.prev.orientation);

guy.VISITED = true;

guy.prev.VISITED = true;

}

}

}

**(c) Verify that your algorithm runs in time O(|W|+|R|), where |W| and |R| denote respectively**

**the lengths of the arrays W and R.**

This solution would take O(n + r) time for the BFS, as per BFS definition, O(n) time to set every wrestler as a Heel or Babyface, and O(r) time to check edges and print fights (if possible).

So overall, we are looking at O(n + r) + O(n) + O(r) = 2O(n + r) = O(n + r).

**(d) Prove that your algorithm is correct.**

Proof by induction, proving base case for a single pair of fighters, say A and B.

Running BFS on A, we get A.d = 0, B.d = 1, B.prev = A, so the code will set A.orientation = Babyface and B.orientation = Heel.

Checking all edges except root of BSF, we look at B and see that B.orientation != B.prev.orientation, so POSSIBLE = true.

Then we run a print command for all nodes !VISITED except root of BFS, so we print:

B (Heel) Vs. A(Babyface), set both their flags to VISITED = 1 and the algorithm repeats, this time with BFS starting at B. By symmetry, the output will be: A (Heel) Vs. B (Babyface)

Induction Step: We assume the algorithm is true for an arbitrary size of fighters and rivalries.

Facts:

* BFS records the previous neighbor
* Distance from any point between 2 adjacent nodes is always different by at most 1, such that id u and v are neighbors, d[u] = d[v] + 1; d[u] = d[v] or d[u] = d[v] - 1

Since we assign orientation based on the parity of the distance from a point, based on the second fact presented, we have 2 possible cases: either adjacent nodes have opposite orientations, if they have a different distance, or they have similar orientation if their distance to the root of BFS was equal. Since we scan all the nodes and look at their adjacency list, we eliminate a possible match if we detect a rivalry (an edge) where both vertices (both fighters) have same orientation. Is a success condition is accepted, by printing every nodes and their prev node, since all paths are unique on BFS, we will print every fight a single time.

**Q4: Let G = (V,E) be a directed graph, in which each vertex v in V is labeled with a unique integer L(v) called the label of v. For each vertex v, let R(v) be the set of vertices that are reachable from v:**

**R(v) = {u in V : there is a path from v to u}**

**Define value(v) to be the minimum label in R(v):**

**value(v) = min{L(u) : u in R(v)}**

**Give an O(|V| + |E|)-time algorithm that computes value(v) for all vertices v in V , that is, your algorithm must print value(v) for each vertex v of G. The graph is presented using the adjacency list data structure. So the input to your algorithm is a pair (n,Adj) where n is the number of vertices in the graph (we take V = {1, 2, . . . , n}), and Adj is an array of length n whose element Adj[v] is the (pointer to the head of the) linked list of neighbors of node v (for 1 <= v <= n). If you need additional data structures (e.g., additional attributes associated with the vertices) clearly describe them.**

Summary:

We will use a modified version of DFS. I addition to the usual information contained in a node, every node will also have a field called value. When running the DFS, we will visit all the nodes reachable from the DFS ‘root’ node, and as we visit reachable nodes, we compare their value with the value stored into value. If the label of the newly visited node is smaller than value, then record it as new value and pass it backwards as a return value from the recursion. By running this altered DFS on all the nodes, we will effectively update the ‘value’ field for every node.

Data structures used for each node x:

value[x]: as per problem definition

Adj[x]: Head pointer to adjacency list of x

prev[x]: pointer to the previous node in the search

visited[x]: Boolean flag to indicate if node x has been visited

computeValue(n, Adj[]){

int i, temp;

// Reset the DFS relevant fields

for(i = 1; i <= n; i++){

visited[i] = false;

prev[i] = NULL;

value[i] = n+1; // Impossible value used as NULL indicator

}

// Visit all vertices at least once with the altered DFS (i counter)

// DFS returns the smallest value we can reach from this node.

for(i = 1; i <= n; i++){

if(!visited[i]){

temp = AlteredDFS(i)

if(temp < value[i])

value[i] = temp;

}

}

// Since we run DFS on all non visited nodes, at this point

// the value field is up to date on all nodes

// So display it

printf("Node: %d, value: %d", i, value[i]);

}

// Following function will run like a DFS, but also match the

// value of nodes visited and compare them with the value stored in

// nodeOfInterest

int AlteredDFS(DFSroot){

int temp, smallest;

// Mark the node as visited

visited[DFSroot] = true;

// Record this node as being the smallest, subject to be overwritten

// by another node in the recursion

smallest = DFSroot;

for each v in Adj[DFSroot] do{

if(!visited[v]){

prev[v] = DFSroot;

// Here is the modified part in DFS

// Either return the smallest value encountered down the DFS path

// or return the value of this node

temp = AlteredDFS(v);

if(temp < smallest)

smallest = temp;

}

}

// Finally return the smallest encountered node

return smallest;

}