Workspace - Data Structures [13/34]

Queues and Stacks

1 - UVa 210 - Concurrency Simulator⚠

- Parsing problem; the main part is just simulation using a deque/list. List is slower, so we're using the deque. Easily TLE...
- TODO!
- **Warning reason: Another parsing problem that I'm usually very afraid of. Got TLE multiple times.**

2 - UVa 514 - Rails ✅

- Beware of input format.

3 - UVa 442 - Matrix Chain Multiplication ✅⚠

- Another parsing problem. We used a a stack of indices that maps to a long array that can hold many temporary results; it might be better if we just make the results a struct and store the structs on the stack.
- **Warning reason: Yet another parsing problem; didn't manage to solve this one using mutual recursion yet. Also, the way I store intermediate results were very messy and were only realized through debugging.**

Linked Lists

4 - UVa 11988 - Broken Keyboard ✅

- Mind the order of insertion; the state (position) is maintained by a pointer to the linked list. Also, when inserting at a iterator to a linked list, the inserted element goes to the back of the current element.

5 - UVa 12657 - Boxes in a Line ✅⚠

- Challenging problem! Requires use of doubly linked list implemented as array (left and right arrays store left and right pointers). The core part is we cannot use O(n) to do a reverse operation. We use 'inv' to mark when an array is inverted and invert the operations 1 to 2, 2 to 1. When collecting the data if 'inv' is turned on we process the sum as :
- Answer = (n*(n+1))/2 - Answer. (If inv==1 && n is even. Take the dual of the current sum as it's inverted)
- **Warning reason: Didn't think of the 'inv' flag; reversing the entire array costs n^2 worst-case time, which gave a TLE. Also didn't think of the post processing that takes care of the summing process.**

Binary trees

6 - UVa 679 - Dropping Balls ✅⚠

- Really good problem. Looking at each ball one by one causes a TLE. Looking at it more carefully, we find that if the number of balls is odd, then the last ball must fall down the left subtree; if it's even, then the ball must fall down the right subtree. Assuming the number is odd, how many balls before the last ball fell down the left subtree? It’s all the other odd-numbered balls, which is (I+1)/2. Assuming the number is even, how many balls before the last ball fell down the right subtree? It’s all the other even balls, (I/2). Then we can just recursively make this judgement and reduce I by half at each level of the tree, until we reach the leaves.

- **Warning reason: The O(h) solution is exceptionally smart! It’s described in Rujia Liu’s book, and I didn’t think of it before.**
- Code:

```
IterativeFallTree(D, I) {
  int k;
  for(int i = 1; i <= D - 1; ++i) { // Stop at leaves level D
    if (I%2) { // if odd
      I = (I+1)/2; k *= 2; // go down left tree
    } else { // if even
      I /= 2; k = 2*k+1; // go down right tree
    }
  }
  return k;
}
```

```
RecursiveFallTree(D, I, k) {
  if (D == 1) {
    return I/2;
  } else {
    if (I%2) return RecursiveFallTree(D-1, (I+1)/2, k*2);
    else return RecursiveFallTree(D-1, I/2, 2*k+1);
  }
}
```

7 - UVa 122 - Trees on the Level ✅

- We need to maintain a dynamic tree structure + do BFS on tree (using a queue). As simple as that. Just beware of the edge cases.

8 - UVa 548 - Tree ✅☆

- The toughest part is the input. Use sstream to parse the integers - it actually saves some time. There’s a crude atoi version that for some reason always causes TLE in the trash/ folder.
- Pass in a reference to an int to store the leaf node value. We can make dual arrays of in order and post order traversal (O(n) preprocessing) in order to make the worst case time of the tree traversal from n*log(n) to log(n)

(just table lookup the position of root in the dual array of in order traversal). We don’t have to explicitly construct the tree, we can just construct the solution along the way using a dynamic-programming-like recurrence. But there’s no repeated leaves (and we proceeded top-down), so we do not memoize.

- Reason for starring: Our algorithm does not explicitly construct the tree, resulting in n+logn runtime instead of n+n*log(n) runtime as described in conventional solutions.

9 - UVa 839 - Not so Mobile ✅

- Model this as a recursive process. Quite a simple problem. Pay attention the alternative to using pairs as a return result is passing in a returned value by reference for modification.

10 - UVa 699 - The Falling Leaves ✅⚠

- The problem setter has unfortunately mistaken what “horizontally” and “vertically” means in English.
- Don’t complicate!!! Do not build the tree! Make a large-enough array, and start collecting leaves (constructing the solution right away) using recursion by starting at the middle of the array and going left/right. Finish by traversing the big array ‘L’ and recording everything positive to a separate vector for processing output (the last member must ‘\n’).
- **Warning reason: On the first try, I actually attempted to build the tree and then count the number of leaves, and got WA. Thought of too many complications.**
- Code:

```
void build(int *L, int T, int d)
{ if (T== -1) return;    // base case; go back
  int l,r;    // left and right
  L[d] += T;    // accumulate here
  cin >> l;    // this is an in-order traversal
  build(L, l, d-1);
  cin >> r;
  build(L, r, d+1);
}
```

Non-binary Trees

11 - UVa 297 - Quadtrees ✅☆⚠

- Warning: be careful and read the problem!! I thought about this problem as a simple bitwise-OR problem first; however, I couldn’t use a bitset because the different levels contribute different weights towards the final result.
- We can use a single recursive call to handle both strings (without explicitly constructing the tree). The idea of the recursion goes as follows: Record a pointer towards each of the two strings representing a pre-order traversal of the trees. At each level, do a case-by-case analysis of the two values. If both are parent nodes (intermediaries towards result); if neither are, manually do an OR and return it; for one intermediary and one terminal, if the terminal is black, then the terminal’s result will have a higher weight over the children of the intermediary, so just evaluate the terminal; to the opposite, if the terminal is white, then we call another recursion “build1” that builds the subtree of the intermediary.
- For skipping intermediaries, we call “build1” without recording the result of the recursion.

- This is similar to parsing CFGs. Only increment the pointers to strings on terminal nodes, and direct expansion on nonterminals via recursion.
- Why can't we model this as backtracking, store a global value, and update it on the leaves? because we also need 'build1' to skip out out-weighted nonterminals, and hence we need to discard intermediary results sometimes. If one were forced to use backtracking, this can be changed by having another tree walker function that is dedicated to skipping intermediaries.
- Star reason: I thought up the "dual-pivot" search myself and finished it correctly without mistakes.
- Warning reason: Again, it bears

```
int build1(string& x, int& px, int level)

{ if (px == x.size()) return 0;

    switch(x[px]) {

        case 'p': {

            ++px;

            int a1 = build1(x, px, level/2);

            int a2 = build1(x, px, level/2);

            int a3 = build1(x, px, level/2);

            int a4 = build1(x, px, level/2);

            return a1+a2+a3+a4; }

        case 'e':

            ++px;

            return 0;

            break;

         case 'f':

            ++px;

            return level*level;

            break;

     }

    return 0;

}


int build(string& x, string& y, int& px, int& py, int level)

{ if (px == x.size() && py == y.size()) return 0;

    if (px == x.size()) return build1(y, py, level);

    else if (py == y.size()) return build1(x, px, level);

    if (x[px] == 'p' && y[py] != 'p') {

        if (y[py] == 'f') {

            ++py;
            build1(x, px, level);
            return level * level;
        } else {
            ++py;
            return build1(x, px, level);
         }
    } else if (x[px] != 'p' && y[py] == 'p') {
        if (x[px] == 'f') {
            ++px;
            build1(y, py, level);
            return level*level;
        } else {
            ++px;
            return build1(y, py, level);
        }
    } else if (x[px] == 'p' && y[py] == 'p'){
        ++px; ++py;
        int a1 = build(x, y, px, py, level/2),
            a2 = build(x, y, px, py, level/2),
            a3 = build(x, y, px, py, level/2),
            a4 = build(x, y, px, py, level/2);
            return a1+a2+a3+a4;
    } else { /* both not parents */
        int p = 0;
        if (x[px] == 'f') p = 1;
        if (y[py] == 'f') p = 1;
        ++px; ++py;
        return p * level * level;
    }
}
```

Graph - DFS

12 - UVa 572 - Oil Deposits ✅

- This is just a classical flood-fill.

- Also learned how to construct an adjacency matrix via flood fill and the register method.

13 - UVa 1103 - Ancient Messages ✅⭐⚠️

- This requires an alternative version of flood fill that also constructs a graph's adjacency list representation.
- There's a solution to do that in O(n^2 * log(n)^2) using BSTs. We'll use C++'s built-in set and map.
- If we don't use 'set' and 'map' we can also either use an integer hash table or write our own BST.
- Our solution here involves first "zooming out" from the input picture; namely, we magnify the binary image from size m*n*4 to (m+2)*(n*4+2). The topmost/bottommost row, leftmost/rightmost column are all blank. This helps us avoid the degeneracy in which the blank space on the outside is being cut off by a very large black space that spans the entire picture.
- After zooming out we use our flood fill to construct the graph. We use an additional BST instance to keep track of the original color of our filled block, i.e. whether if it is black or white.
- After constructing the graph we traverse each node (which is a connected region on the picture), and if the color of the node is originally black, we determine which letter it is based on the number of adjacencies it has (each character has different number of inner spaces, which helps us determine the overall size).
- Lesson: Life would be very hard (very, very hard) without C++ STL!!
- Star reason: I thought up the procedure that builds the adjacency list from the flood fill and the "zoom-out" preprocessing to avoid degeneracies.
- Warning reason: Very intricate problem; the biggest hint is to count how many "blank spaces" that are contained within a character.
- Code for the described parts:

```
const char* Hash = "0WAKJSD";


struct graph {
 map<int, set<int> > List;
 map<int, short> orig;
 void add_edge(int s, int t) {
  this->List[s].insert(t);
  this->List[t].insert(s);
 }
 void set_orig(int s, short s1) {
  this->orig[s] = s1;
 }
 int size;
 graph() { this->size = 0; }
};


// … …
```

```
void build(int* M, graph& G, int m, int n, int i, int j, int t, int r)
{
 if (i >= m || j >= n || i < 0 || j < 0) return;
 if (M[i*n+j] != t) {
  if (M[i*n+j] > 1 && M[i*n+j] != r) {
   G.add_edge(r, M[i*n+j]);
   G.set_orig(r, t);
  }
  return;
 }
 M[i*n+j] = r;
 build(M, G, m, n, i, j+1, t, r);
 build(M, G, m, n, i, j-1, t, r);
 build(M, G, m, n, i+1, j, t, r);
 build(M, G, m, n, i-1, j, t, r);
}

// … …

int main()
{
// … …
    for(int i = 0; i < m; ++i)
        for(int j = 0; j < n; ++j)
            if (mat[i*n+j] < 2) {
                build(mat, g, m, n, i, j, mat[i*n+j], reg);
         ++reg; ++g.size;
     }
 printf("Case %d: ", cnt);
 vector<char> output;
 for(const pair<int,set<int> >& pk : g.List) {
     if (g.orig[pk.first])
  output.push_back(Hash[pk.second.size()]);
 }
 sort(output.begin(), output.end());
 for(const char& c : output) printf("%c", c);
 printf("\n");
// ……
}
```

Graph - BFS

14 - UVa 816 - Abbott’s Revenge ✅⚠️

- Quite complicated BFS, hard to get it right
- Note: in row-column setting (row, col)=(y, x)
- A node is symbolized by the 3-tuple (x, y, dir), where the direction can be N, E, S, W.
- How to transition? Make arrays 'dy' and 'dx' that turns N, E, S, W clockwise or counterclockwise.
- HUGE issue: when starting the BFS DO NOT start at the entry of the maze; this leads to all sorts of problems when dealing with Euler cycles. Mark the next node that proceeds from the entry as the point to start the BFS, and print out the entry to the maze as the last member later.

Graph - Topological sorting

15 - UVa 10305 - Ordering Tasks

Graph - Euler cycle

16 - UVa 10129 - Play on words

Additional Practice - Example Problems

17 - UVa 10562 - Undraw the trees

18 - UVa 12171 - Sculpture

19 - UVa 1572 - Self-assembly

20 - UVa 1599 - Ideal Path

21 - UVa 506 - System Dependencies

22 - UVa 11853 - Paintball

Additional Practice - Practice Problems

23 - UVa 673

24 - UVa 712

25 - UVa 536

26 - UVa 439

27 - UVa 1600

28 - UVa 12166

29 - UVa 804

30 - UVa 806

31 - UVa 127

32 - UVa 246

33 - UVa 10410

34 - UVa 810

35 - UVa 215

36 - UVa 12118