Mcpc18 prep #2

Link to contest: https://open.kattis.com/contests/xkavbj/problems

First step

As always, the first thing to do is read the problem statement carefully and understand what the problem is asking. Read multiple times to make sure you don't miss important details.

Analysis of problem #1 Fire

Input to problem

> A grid of characters representing rooms/walls/fire start locations (up to 1000 x 1000)
> Rules of how fire spreads and how agent moves

Output

> Minimum time to escape grid without catching fire

Working out a solution

Observation: regardless of how you move in the grid, fire spreads in a predictable manner. More precisely, the fire spreads in the grid in a Breadth First manner. You should be familiar with how the search frontier progresses in BFS/DFS, and other graph search algorithms: this is key to coming up with solutions for problems, it is not sufficient to remember how to code DFS or BFS etc. you should understand how the search frontier evolves because this is what makes the algorithms correct.

Now we can separate the problems:

1. First, let's compute for each cell the minimum time required for the fire to reach it. We do this by using BFS
2. We run a second BFS to move in the grid, a cell is considered safe if the following conditions are met:
    a. Cell is within the grid (cannot go off the edges)
    b. Cell is not a wall

c. Time to reach cell is strictly smaller than time taken by fire to reach this cell (otherwise agent will catch fire)

Stopping condition is when agent reaches one of the 4 borders of the grid.

Link to solution (cpp): https://ideone.com/eKUO3o

Analysis of problem #2 Font

Input to problem

Array of size N (up to 25)
Each element is a string of lowercase characters (up to 100 strings)
Strings are distinct so input is a set of strings

Output

Want to compute the number of subsets of the set of strings so that:

1. Each character of the alphabet is contained in at least one of the strings in the subset

Working out a solution

First attempt

First thing that comes to mind is to try to generate all of the possible subsets (there are 2^25 subsets in total, ~ 33 x 10^7) and then check each subset and update the answer. This solution will be described in the alternative solution, but if not implemented carefully, it might not pass the time limit.

Second attempt

Problem does not ask us to enumerate the subsets but to count them. When a problem asks you to count configurations then it is likely that there are faster ways to count (bruteforce is always a final thing we can try). Example: to count the number of subsets of some set A, you don't write down those subsets and increment a counter each time, you know that the answer is 2^(size of A) because you have seen a proof or you remember this from some lecture.

The time limit is 1 second so we can try to optimize things a little bit before submitting.

First thing to do is to represent the strings as binary masks. There are 26 unique characters at most in any of the strings so we can use a binary number to flag the characters which are

present in the string). Example the string “acg” will be represented by the number: 00000000000000000001000101. Each digit tells us if the character is present (1) or not (0).

When you try to count stuff, if you don’t have a closed formula then it’s a good idea to consider dynamic programming (it might or might not work but it does not hurt to consider it). The state is (pos, mask), it means that we are at position pos and we have seen the characters present in mask.

A stopping condition is when mask has all of the characters flagged. This is a second optimization because we don’t have to traverse the strings til the end if we have seen all of the characters. We know that any subset of the remaining strings can be added without changing the mask so we can just add 2^(number of remaining strings) to the answer

There are two transitions out of each state:

1. Move to the next position without considering the current string: (pos, mask) -> (pos + 1, mask)
2. Add the current string to the subset and update mask: (pos, mask) -> (pos + 1, mask | B) where B is the binary representation of the string sitting at pos

For this problem it makes sense to have a top down approach to avoid generating all of the useless states which might exist.

Also it’s better to use a hash table to store the states for the same reason as above. Your compiler might catch that the size of the memo is too big if you try to use a static array.

Link to solution (cpp): https://ideone.com/mseunO

Alternative solution: Problem #2 (Font)

Now let’s describe the 2^n solution we talked about briefly in the actual analysis of this problem.

First, like in the previous solution, let’s generate an array “mask”, where mask[i] = the mask of characters in the i-th string (bit “j” is set if the j-th bit of the alphabet appears in the string).

Now we generate all the subset of the strings which are 2^n in total (since there’s “n” string). We’ll do this using bit manipulation & iterating from 0 to 2^n.

The problem here is that for each mask of a particular subset we’ll do a O(n) to compute the bitwise OR in order to obtain the final mask.

Now the complexity is O(n * 2^n) instead of O(2^n). Which is not as efficient. We definitely need to get rid of the “n” factor.

To do so, we'll need to compute the bitwise OR in constant time O(1).

The idea is to compute the bitwise OR of all the subset of the first half of the array (there are 2^(n/2) such subset, around 2^13 = ~8192). Let's store these values for all masks in an array and call it "orsFirst" where orsFirst[mask] is the the bitwise OR of the elements in the first half of the array which are "set" in the mask.
Same goes for the other half. ("orsSecond")

Now, let's say you have a mask "m", how do we use these two array to compute the final bitwise OR in O(1) time?
We simply split it to 2 halves, we get the value of the first half from the "orsFirst" array and the value of the second half from the "orsSecond" array. The bitwise OR of these two values is what we search for.

Example of this process: let's say I have an array of size 6, and I want the bitwise OR of the mask "101001", we'll take the value of the first half of the mask (which is "101") from the first array, and do the bitwise OR with the second half of the mask (which is "001")

CPP Code for more details: https://ideone.com/X1VpU0

Conclusions

Dynamic programming (with optimization tricks and sometimes bit tricks) is famous. It is good to know these techniques and be confident with bit manipulation and state representation. Graph theory is very rich in terms of problems and algorithms. You must be able to represent graphs and write algorithms that traverse them. DFS and BFS are must know algorithms as well. Whenever you try to optimize shortest distances in a graph structure consider BFS, the state is not always a plain node, it might be a combination of objects.

Cheers