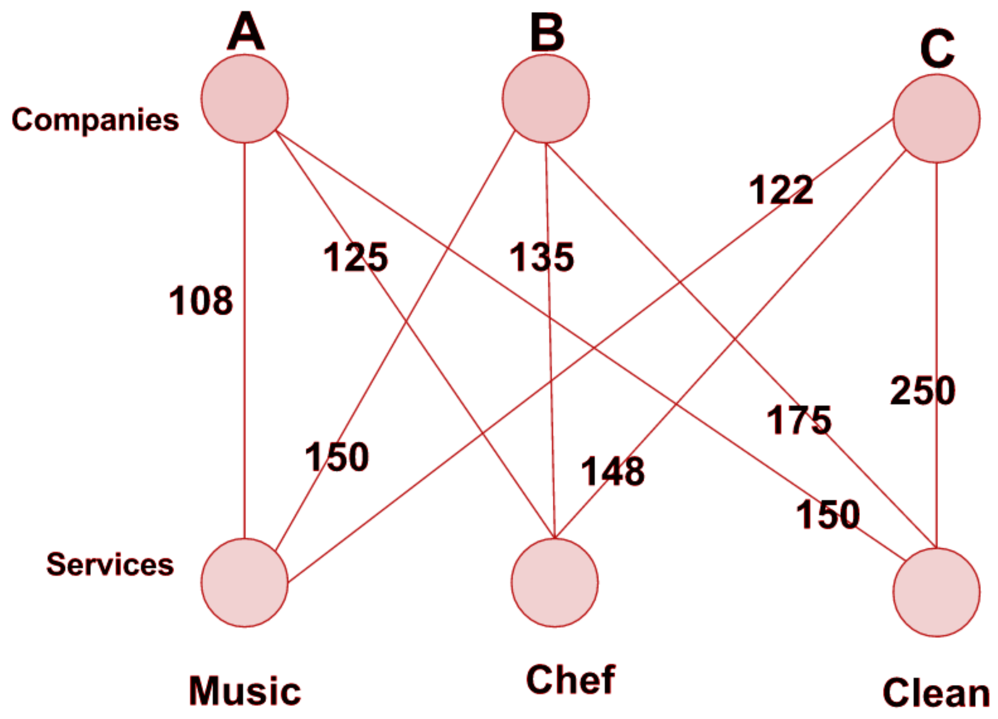
DP with Bitmasking

We can often pair up bit masking with DP to solve problems in exponential time which would otherwise require time complexity of the order of O(N!) for any given N. A small and easy trick to judge if a problem can be solved by bit masking DP is that the constraints are small and often N<=20 where N is the size of the set we want to store as a state.

Let us look at an example that uses this concept.

Assignment Problem

Problem Statement: There are N companies and N services, each service is to be alloted to a single company. We are also given a matrix cost of size N×N, where cost[i][j] denotes how much company i is going to charge for service j. Now we need to assign each service to a company in such a way that the total cost is minimum. Note that each service is to be alloted to a single company, and each company will be allotted exactly one service.



In the above figure the edge weight denotes the cost of service between a company (A, B, C) and a service (Music, Chef, Clean).

Naive Approach: Let us try to see the naive way to solve this problem. Since there are N companies and only that many services, we see that optimal assignment will be some permutation of the services to the companies. So we can enumerate all possible orders of services and then take the order that minimizes our total cost.
The code for the above approach is as follows:

```
function assignmentProblem(cost, N)
        /*
                We are given the cost matrix
                and the number of companies
        */

        assignment = array[n]           // array to store the assignment permutation
        for i = 0 to N
                assignment[i] = i        // assigning task i to person i

        //   final result
        res = INFINITY

        for j = 0 to factorial(N)
                //   current cost of the assignment
                total_cost = 0
                for i = 0 to N
                        total_cost = total_cost + cost[i][assignment[i]]
                res = min(res, total_cost)
                generate_next_greater_permutation(assignment)

        return res
```

Time Complexity O(N!*N), where N is the number of companies. Since we are iterating over all the N! permutations of the service and each time we are looping over the services to calculate the cost, the total time complexity turns out to be O(N!*N).

Space Complexity O(1), Since we are only using constant additional space to store the cost of an assignment, the total space complexity becomes O(1).

Dynamic Programming Approach: Let's try to improve it using dynamic programming. Suppose the state of dp is *(k,mask)*, where *k* represents that company 0 to k−1 have been assigned a service, and mask is a binary number, whose ith bit represents if the ith service has been assigned or not. Now, suppose, we have *answer(k,mask)*, we can assign a service i to company k, if ith service is not yet assigned to any company i.e. *(mask & (1<<i) == 0)* then, *answer(k+1,mask|(1<<i))* will be given as:

answer(k+1,mask|(1<<i)) = min(answer(k+1,mask|(1<<i)),answer(k,mask) + cost[k][i])

One thing to note here is k is always equal to the number set bits in the mask, so we can remove that. So the dp state now is just (mask), and if we have answer(mask), then

answer(mask|(1<<i)) = min(answer(mask|(1<<i)), answer(mask)+cost[x][i])

here x=number of set bits in mask.

Complete Code is given below:

```
function assignmentProblem(cost, N)
    /*
        We are given the cost matrix
        and the number of companies
    */

    //   dp array stores the cost for a particular subset denoted by the mask i
    dp = array[power(2, N)]

    for i = 0 to power(2,N) - 1
        dp[i] = INFINITY

    //   base case as empty subset has cost 0
    dp[0] = 0

    //   iterate over all possible mask
    for mask = 0 to power(2, N) - 1
        x = count_set_bits(mask)
        for j = 0 to N - 1
            if jth bit is not set in i
                //   recurrence relation of dp
                dp[mask|(1<<j)] = min(dp[mask|(1<<j)], dp[mask]+cost[x][j])

    //   return the cost of the whole set
    return dp[power(2,N)-1]
```

.

Time Complexity O((2^N)*N), where N is the number of companies. Since we are iterating over all the 2^N subsets of the service and each time we are looping over the services to calculate the cost, the total time complexity turns out to be O((2^N)*N).

Space Complexity O(2^N), where N is the number of companies. Since we are creating a dp array of size 2^N, the total space complexity is O(2^N).

Applications of bit masking

- Since counting bits, and checking, setting and unsetting a bit in a bitmask is comparatively faster than a boolean array, this technique is widely used to speed up optimization algorithms if the number of elements are small in the subset
- It is also used to solve popular algorithmic problems like travelling salesman, number of hamiltonian cycles, job assignment problem etc.
- It can be used to make a recursive algorithm iterative, for eg. generating all subsets and hence reducing the overhead memory cost associated with recursive memory stack.