Competitive Programming and Mathematics Society

Advanced Dynamic Programming

Advanced DP

Programming Committee

Table of Contents

UNSW CPMSoc

What is Depth First Search (DFS)?

- Traversal technique used to visit all nodes of the tree
- Depth-first strategy to explore as far along each branch as possible before backtracking
- Time complexity is $\mathcal{O}(V + E)$

DFS implementation

UNSW CPMSoc

https://thealgoristsblob.blob.core.windows.net/thealgoristsimages/dfs.gif

```
vector<int> adj[100005]; // Adjacency list representation
bool seen[100005];       // Array to keep track of visited nodes

void dfs(int node) {
    seen[node] = true; // Mark the current node as visited

    // Iterate through all adjacent nodes of the current node
    for (int i = 0; i < adj[node].size(); i++) {
        int adjacentNode = adj[node][i];
        // If the adjacent node hasn't been visited, recursively call dfs on it
        if (!seen[adjacentNode]) {
            dfs(adjacentNode);
        }
    }
}
```

DP On Trees

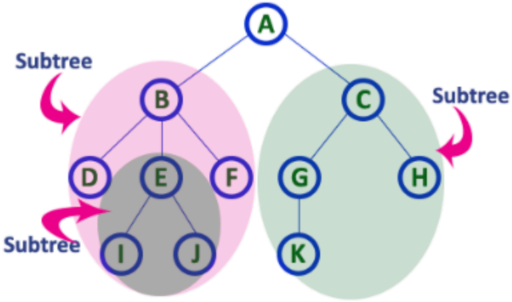
UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

As with regular DP, we're trying to break a problem down into subproblems. When solving tree problems, we will often build up our solution by combining information from the same problem applied to subtrees.

DP On Trees

As with regular DP, we're trying to break a problem down into subproblems. When solving tree problems, we will often build up our solution by combining information from the same problem applied to subtrees.



Framework

1. Pick a node to root your tree at (this can be arbitrarily done).

Framework

1. Pick a node to root your tree at (this can be arbitrarily done).
2. Define how to solve the given problem by combining solutions to the same problem from the subtrees of the root.
   - This is where the DP magic happens, as you express how the solution at a given node depends on the solutions of its subproblems.
   - Eg. to find the largest value in a tree, we can take the maximum value out of the current root's value and the max value in all its subtrees (ie. solving the same problem for the root's subtrees).

Framework

UNSW CPMSoc

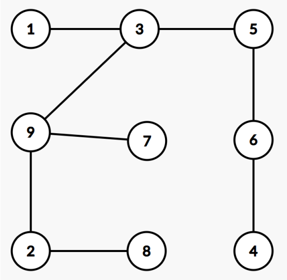
Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

1. Pick a node to root your tree at (this can be arbitrarily done).
2. Define how to solve the given problem by combining solutions to the same problem from the subtrees of the root.
   - This is where the DP magic happens, as you express how the solution at a given node depends on the solutions of its subproblems.
   - Eg. to find the largest value in a tree, we can take the maximum value out of the current root's value and the max value in all its subtrees (ie. solving the same problem for the root's subtrees).
3. Determine your base cases
   - Usually a leaf node (or a subtree with a single node). The max value in this tree is trivially the only value.

Let's solve a problem

UNSW CPMSoc

You are given a tree consisting of n nodes. Your task is to determine the diameter of the tree. The diameter of a tree is the maximum distance between two nodes.



Here we have a tree with diameter 6, which is provided by the path from 8 to 4.

Our Thought Process

- Let us consider the tree rooted at an arbitrary node.
  - How can we break the problem down?

Our Thought Process

- Let us consider the tree rooted at an arbitrary node.
  - How can we break the problem down?
- We observe that the longest (simple) path in our tree (diameter) can either pass through our root or not.

1. **Does not pass through the root**

Our Thought Process

- Let us consider the tree rooted at an arbitrary node.
  - How can we break the problem down?
- We observe that the longest (simple) path in our tree (diameter) can either pass through our root or not.

1. **Does not pass through the root**
   - In this case our longest path in our tree will be the longest of the paths entirely within each of our root's subtrees.
2. **Does pass through the root**

Our Thought Process

- Let us consider the tree rooted at an arbitrary node.
  - How can we break the problem down?
- We observe that the longest (simple) path in our tree (diameter) can either pass through our root or not.

1. **Does not pass through the root**
   - In this case our longest path in our tree will be the longest of the paths entirely within each of our root's subtrees.
2. **Does pass through the root**
   - Then our longest path will be the concatenation of the two longest paths within subtrees that start at the root of their respective subtrees.
   - These longest paths within the subtrees that start at the root are essentially the heights of the subtrees.

- Now our base cases are simply leaf nodes (ie. subtrees with one node) which clearly have a diameter of 0 and a height of 0.

Let's try and implement!

Figure: https://vjudge.net/contest/620660

Implementation

UNSW CPMSoc

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
using namespace std;

int n;
vector <vector<int>> edges; // adjacency list
vector <int> parent; // the parent of each node (determined by dfs)

vector <int> dp; // the longest path in the subtree rooted at each node
vector <int> height; // the height of each nod

void subtree_diameter(int i);
int root;
```

Moar implementation

```
int main(void) {
    cin >> n;
    // resize everything
    edges.resize(n + 5);
    dp.resize(n + 5,-1);
    height.resize(n + 5, -1);
    parent.resize(n + 5, -1);

    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    root = 1; // arbitrary root
    subtree_diameter(root); // run algorithm on root node
    cout << dp[root]; // print answer
    return 0;
```

And again :qiqifallen:

```
//calculates the diameter of the subtree rooted at i
void subtree_diameter(int i) {
    //base case: is a leaf in the rooted tree
    if (edges[i].size() <= 1 && i != root) {
        height[i] = 0;
        dp[i] = 0;
        return;
    }

    int height_of_tallest_child = -1;
    int height_of_2nd_tallest_child = -1;
    int max_child_diameter = 0;
```

Implementation again

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

```
    for (int j: edges[i]) {
        if (j == parent[i]) continue;
        parent[j] = i; // make the node that we just came from the parent

        subtree_diameter(j); // recurse on each child

        // update the values for the 2 tallest children and longest path contained withi
        if (height[j] > height_of_tallest_child) {
            height_of_2nd_tallest_child = height_of_tallest_child;
            height_of_tallest_child = height[j];
        } else {
            height_of_2nd_tallest_child = max(height_of_2nd_tallest_child, height[j]);
        }
        max_child_diameter = max(max_child_diameter, dp[j]);
    }
```

And again :qiqifallen:

```
        // the height of the subtree rooted at i
        height[i] = height_of_tallest_child + 1;

        // the longest path in the subtree if it doesnt include the root of the subtree
        int excl_i = max_child_diameter;

        // the longest path in the subtree if it does include the root node
        int incl_i = (height_of_tallest_child + 1) + (height_of_2nd_tallest_child + 1);
        // the longest path contained within the subtree overall.
        dp[i] = max(incl_i, excl_i);
        return;
    }
```

Binary Representation

- In computer systems, we typically store numbers in their binary representation.

Binary Representation

- In computer systems, we typically store numbers in their binary representation.
- Binary numbers use base 2, with digits 0 and 1.

Binary Representation

- In computer systems, we typically store numbers in their binary representation.
- Binary numbers use base 2, with digits 0 and 1.
- To find the number represented by a sequence of binary digits we multiply each digit by the appropriate power of 2 and add up the results. In general, the value of an n-bit sequence

$$b_{n-1}...b_1b_{0[2]} = b_{n-1}2^{n-1} + \cdots + b_1 2^1 + b_0 2^0 = \sum_{i=0}^{n-1} b_i 2^i$$

Binary Representation

- In computer systems, we typically store numbers in their binary representation.
- Binary numbers use base 2, with digits 0 and 1.
- To find the number represented by a sequence of binary digits we multiply each digit by the appropriate power of 2 and add up the results. In general, the value of an n-bit sequence

$$b_{n-1}...b_1b_{0[2]} = b_{n-1}2^{n-1} + \cdots + b_1 2^1 + b_0 2^0 = \sum_{i=0}^{n-1} b_i 2^i$$

- For example, $10011_{[2]}$ represents $1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 16 + 2 + 1 = 19.$

Binary Representation

- Similarly, $1000100101_{[2]}$ is represented by

512	256	128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	1	0	1

so that $1000100101_{[2]} = (1 * 1) + (1 * 4) + (1 * 32) + (1 * 512) = 1 + 4 + 32 + 512 = 549$.

Bitwise Operators

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

- You likely already know basic logical operations like AND and OR. Using

```
if(condition1 && condition2)
```

checks if both conditions are true, while

```
if(c1 || c2)
```

requires at least one condition to be true.

Bitwise Operators

- You likely already know basic logical operations like AND and OR. Using

```
if(condition1 && condition2)
```

  checks if both conditions are true, while

```
if(c1 || c2)
```

  requires at least one condition to be true.
- Same can be done bit-per-bit with whole numbers, and it's called bitwise operations.

Bitwise Operators

UNSW CPMSoc

- The bitwise **NOT**, or bitwise complement, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value.

```
int x = 8; // 0111 in binary
x = ~x;    // 1000 in binary
```

- The bitwise **AND** is a binary operation that takes two binary representations and performs the logical AND operation on each pair of the corresponding bits.

```
int x = 5;      // 0101 in binary
int y = 3;      // 0011 in binary
int z = x & y;  // 0001 in binary
```

Bitwise Operators

- The bitwise **OR** is a binary operation that takes two binary representations and performs the logical inclusive OR operation on each pair of corresponding bits.

```
int x = 5;      // 0101 in binary
int y = 3;      // 0011 in binary
int z = x | y;  // 0111 in binary
```

- The bitwise **XOR** is a binary operation that takes two binary representations and performs the logical exclusive OR operation on each pair of corresponding bits.

```
int x = 5;      // 0101 in binary
int y = 3;      // 0011 in binary
int z = x ^ y;  // 0110 in binary
```

Bitwise Operators

- Left Shift $<<$: This operation moves all the bits in a binary number to the left by a specified number of positions.
- Right Shift $>>$: This operation moves all the bits in a binary number to the right by a specified number of positions.

Operator	Example	Result
`a << b` Left Shift b bits `a >> b` Right Shift b bits	00110101 << 1	01101010
	10110001 >> 1	01011000

Bitwise Operators

- In Bitmask DP, we will need the following operations
- To test if a bit n is set in x:

```
if (x & (1 << n)) {


}
```

Bitwise Operators

- In Bitmask DP, we will need the following operations
- To test if a bit n is set in x:

```
if (x & (1 << n)) {

}
```

- To set a bit n in x:

```
x = x | (1 << n);
```

Bitwise Operators

- In Bitmask DP, we will need the following operations
- To test if a bit n is set in x:

```
if (x & (1 << n)) {

}
```

- To set a bit n in x:

```
x = x | (1 << n);
```

- To clear a bit n in x:

```
x = x & ~(1 << n);
```

Bitmask DP

- Bitmask DP is one common technique to solve **intractable** problems.
- **Intractable** problems are problems that can be solved in theory but in practice, take too long for their solution to be useful.

Bitmask DP

- Bitmask DP is one common technique to solve **intractable** problems.
- **Intractable** problems are problems that can be solved in theory but in practice, take too long for their solution to be useful.
- The best-known solutions for intractable problems generally run in exponential or subexponential time.

Bitmask DP

- Bitmask DP is one common technique to solve **intractable** problems.
- **Intractable** problems are problems that can be solved in theory but in practice, take too long for their solution to be useful.
- The best-known solutions for intractable problems generally run in exponential or subexponential time.
- Some examples of intractable problems are
  - **Subset sum**: Given a set of integers, is there any subset whose sum is 0?
  - **Hamiltonian path**: Given a graph, does a Hamiltonian path exist?
  - **Travelling salesman**: Given a graph, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Travelling Salesman

UNSW CPMSoc

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

Travelling Salesman

UNSW CPMSoc

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Brute force?

Travelling Salesman

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Brute force?
- Since we are given a weighted, complete graph, we can simply try every single route from the starting city and calculate the cost of the route, then take the minimum cost route we encounter.

Travelling Salesman

UNSW CPMSoc

> There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Brute force?
- Since we are given a weighted, complete graph, we can simply try every single route from the starting city and calculate the cost of the route, then take the minimum cost route we encounter.
- Time complexity?

Travelling Salesman

UNSW CPMSoc

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Brute force?
- Since we are given a weighted, complete graph, we can simply try every single route from the starting city and calculate the cost of the route, then take the minimum cost route we encounter.
- Time complexity?
- Since there are a total of $N!$ different routes which we could have taken, the total time complexity is $O(N!)$. Unfortunately, this is too slow to pass :(

Travelling Salesman

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Assume we are comparing two different ways to go from City A to City B, both of which visit the same intermediate cities but in a different order.
- Logically, whichever one of these two paths is shorter will always be better than the other, and will always be the preferred path to take.
- Therefore, there is no reason to continue adding cities onto the longer path. Unlike the naive solution, the dynamic programming solution for this problem takes advantage of this

Travelling Salesman

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

> There are N cities $(2 \leq N < 20)$. Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Let's think of a possible sub-problem that we can reuse to build up to the full solution.

Travelling Salesman

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

There are N cities $(2 \leq N < 20)$. Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Let's think of a possible sub-problem that we can reuse to build up to the full solution.
- Let $dp[S][j]$ represent the shortest path that starts from vertex 1, visits every single city in S and ends at city j.

Travelling Salesman

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Let's think of a possible sub-problem that we can reuse to build up to the full solution.
- Let $dp[S][j]$ represent the shortest path that starts from vertex 1, visits every single city in S and ends at city j.
- The recurrence can then be formulated as

$$dp[S][j] = \min_{u \in S}(dp[S \setminus \{u\}][u] + dist[u][j]).$$

Travelling Salesman

UNSW CPMSoc

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Note that to represent S in our implementation, we will use our previously discussed bitwise tricks. We represent S with a number where if the i-th least significant bit of the number is set, it represents that city i is in the set.

Travelling Salesman

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Note that to represent S in our implementation, we will use our previously discussed bitwise tricks. We represent S with a number where if the i-th least significant bit of the number is set, it represents that city i is in the set.
- E.g., the number $11 = 1011_{[2]}$ represents we have visited cities 0, 1 and 3. Note that it might be more convenient to then 0-index our cities.

Travelling Salesman

UNSW CPMSoc

> There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Note that to represent S in our implementation, we will use our previously discussed bitwise tricks. We represent S with a number where if the i-th least significant bit of the number is set, it represents that city i is in the set.
- E.g., the number $11 = 1011_{[2]}$ represents we have visited cities 0, 1 and 3. Note that it might be more convenient to then 0-index our cities.
- The total number of subproblems is simply the size of our dp array which is $dp[S][j]$, where $S \leq 2^n$ and $j \leq n$, so we have 2^n subproblems.

Travelling Salesman

UNSW CPMSoc

There are N cities ($2 \leq N < 20$). Given the distance between each pair of cities, find the shortest possible path that visits every city and returns to the origin city (city 1).

- Note that to represent S in our implementation, we will use our previously discussed bitwise tricks. We represent S with a number where if the i-th least significant bit of the number is set, it represents that city i is in the set.
- E.g., the number $11 = 1011_{[2]}$ represents we have visited cities 0, 1 and 3. Note that it might be more convenient to then 0-index our cities.
- The total number of subproblems is simply the size of our dp array which is $dp[S][j]$, where $S \leq 2^n$ and $j \leq n$, so we have 2^n subproblems.
- Each subproblem takes n iterations of a for loop to solve, so the total time complexity is $O(n^2 2^n)$.

Implementation

```
int tsp(int mask, int cur) {
    if (mask == (1 << n) - 1) {
        // now we go from node cur -> node 0
        return adj[cur][0];
    }

    if (dp[mask][cur] != -1) return dp[mask][cur];

    int ans = 1e9;

    for (int v = 0; v < n; v++) {
        if (!(mask & (1 << v))) { // this node is unvisited
            int cur = adj[cur][v] + tsp(mask | (1 << v), v);
            ans = min(ans, cur);
        }
    }
    return dp[mask][cur] = ans;
}
```

Elevator Rides problem

Problem statement: There are n people who want to get to the top of a building which has only one elevator. You know the weight of each person w_i and the maximum allowed weight in the elevator x. What is the minimum number of elevator rides?

Elevator Rides problem

Problem statement: There are n people who want to get to the top of a building which has only one elevator. You know the weight of each person w_i and the maximum allowed weight in the elevator x. What is the minimum number of elevator rides?

Constraints: $1 \leq n \leq 20$, $1 \leq x \leq 10^9$, $1 \leq w_i \leq x$

Example:
If $n = 4, x = 10$ and the four people's weights were: $4, 8, 6, 1$

Elevator Rides problem

Problem statement: There are n people who want to get to the top of a building which has only one elevator. You know the weight of each person w_i and the maximum allowed weight in the elevator x. What is the minimum number of elevator rides?

Constraints: $1 \leq n \leq 20$, $1 \leq x \leq 10^9$, $1 \leq w_i \leq x$

Example:
If $n = 4, x = 10$ and the four people's weights were: $4, 8, 6, 1$

In this case here, we'll put $4, 6$ in one elevator and $1, 8$ in another elevator, so our program should return the number of elevators we used = 2.

Approach 1 - Greedy

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

A possible idea we may have (at least what I had as a first thought) is that we can keep putting in the heaviest people into each elevator until we cannot fit anymore people in which case we add 1 to our answer and start on a new elevator.

Try on example test case:

Example: If $n = 4, x = 10$ and the four people's weights were: $4, 8, 6, 1$

Ok this seems to work!

Approach 1 - Greedy

A possible idea we may have (at least what I had as a first thought) is that we can keep putting in the heaviest people into each elevator until we cannot fit anymore people in which case we add 1 to our answer and start on a new elevator.

Try on example test case:
Example: If $n = 4, x = 10$ and the four people's weights were: $4, 8, 6, 1$
Ok this seems to work!

Until it doesn't...
If we have this test case: $n = 7, x = 10$ where people's weights are: $6, 3, 3, 2, 2, 2, 2$

Greedy solution: will choose to put $(6, 3), (3, 2, 2, 2), (2)$ which is 3 groups
Optimal solution: will get $(6, 2, 2), (3, 3, 2, 2)$ which is only 2 groups. Therefore we are in serious trouble!

Approach 2 - Brute force

Often when elegant solutions don't work out we turn to our old friend, brute force! Any ideas for how we can attack this with brute force?

Approach 2 - Brute force

Often when elegant solutions don't work out we turn to our old friend, brute force! Any ideas for how we can attack this with brute force?

We can check every order people can stand in and run a simulation of putting people into elevators in that order. From there we just choose the smallest number of elevators. Gotta love the brute-force strategy!!

Approach 2 - Brute force

Often when elegant solutions don't work out we turn to our old friend, brute force! Any ideas for how we can attack this with brute force?

We can check every order people can stand in and run a simulation of putting people into elevators in that order. From there we just choose the smallest number of elevators. Gotta love the brute-force strategy!!

Unfortunately, no algorithm is perfect:(
When we analyze the complexity, we find $O(n! * n)$ because we need to generate every permutation $n!$ of them, and simulate each one $O(n)$

As a rule of thumb, if we need to use permutations, the max n can be is 11. Because $11! = 39,916,800$. Any guesses for what $20! * 20$ is?

Approach 2 - Brute force

Often when elegant solutions don't work out we turn to our old friend, brute force! Any ideas for how we can attack this with brute force?

We can check every order people can stand in and run a simulation of putting people into elevators in that order. From there we just choose the smallest number of elevators. Gotta love the brute-force strategy!!

Unfortunately, no algorithm is perfect:(
When we analyze the complexity, we find $O(n! * n)$ because we need to generate every permutation $n!$ of them, and simulate each one $O(n)$

As a rule of thumb, if we need to use permutations, the max n can be is 11. Because $11! = 39,916,800$. Any guesses for what $20! * 20$ is?

It's ahhh... this number... $48,658,040,163,532,800,000$. Not gonna work in a million years!

Solution - DP!

A crucial step in solving a DP problem is to identify what are the things we actually need in order to solve the problem. In other words, the **states** / **sub-problems**. Often times if we analyze our brute-force solution, we can identify redundancies.

Solution - DP!

A crucial step in solving a DP problem is to identify what are the things we actually need in order to solve the problem. In other words, the **states / sub-problems**. Often times if we analyze our brute-force solution, we can identify redundancies.

Observation: We don't care about the ordering!

If we are deciding who's the ith person in the optimal "elevator-entering order" and we're given: the subset of people that were in the first $i - 1$ places. We only care if there exists an ordering that achieves the following, not the actual ordering itself.

1. Minimizes the number of elevators that we've used for the first $i - 1$ people
   - Because we need to make an optimal choice in which we prefer smaller elevator counts.
2. Maximizes space in the last elevator
   - For two solution options with the same elevator counts, we want the one with more space in the last elevator it used. Because then maybe we can fit this ith person in.

Sub-problem definition

In this case, the information stored by **number of elevators used** and **space in the last elevator** replaced the need to know how the first $i - 1$ people are ordered.

With our sub-problem definition being:
$dp[subset\ of\ i-1\ people]\ stores\ a\ pair(min\ elevators, max\ space\ in\ last\ elevator)$

To represent this state of a subset, we use a bitmask of length n where each on-bit represent people in the subset, and each off-bit represent people not in the subset.

Sub-problem definition

UNSW CPMSoc

In this case, the information stored by **number of elevators used** and **space in the last elevator** replaced the need to know how the first $i-1$ people are ordered.

With our sub-problem definition being:
$dp[subset\ of\ i-1\ people]\ stores\ a\ pair(min\ elevators, max\ space\ in\ last\ elevator)$

To represent this state of a subset, we use a bitmask of length n where each on-bit represent people in the subset, and each off-bit represent people not in the subset.

Complexity: In total we would have 2^n possible dp states and each calculation will require $O(n)$ thus our time complexity have now become $O(2^n * n)$. Much better!
Tip: for problems which brute-force solution has factorial time complexity, try think of using bitmask DP to turn into exponential complexity.

Transition + Implementation

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

```
pair<int,int> dp[1<<N];
int n,w[N],x;

pair<int,int> solve(int s){
    if(dp[s].first!=INF) return dp[s];

    for(int i=0;i<n;i++){
        if(s & (1<<i)){
            auto t = solve(s-(1<<i));
            int min_last,min_number;
            if(t.second+w[i]<=x) {
                min_last = t.second+w[i];
                min_number = t.first;
            }
            else{
                min_last = min(t.second,w[i]);
                min_number = t.first+1;
            }
            pair<int,int> temp = {min_number,min_last};
            dp[s] = min(dp[s],temp);
        }
    }
    return dp[s];
}
```

Further events

Please join us for:

- Math Royale (next Thursday 1pm)
- Utilising C++ to tackle coding interviews (W3 Friday)

All details are on our facebook, discord and instagram!