1 Bitmask Dynamic Programming

1.1 Introduction to DP

A lot of ways understand what dynamic programming is.

Note: Bellman quote *(Bellman, 1953) An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions.*

The quote above sounds complicated, but it really just states the best answer must be made from a series of optimal choices. You can break the problem down into individual steps and at each step, the solution is making the best choice.

1. **Super Strong Induction** Assuming that my starting is optimal, at an arbitrary step k, i can assume all decisions made prior to k are optimal. This is also where you see optimizations in speed by recalling previous calculations.

2. **Not greedy** Greedy algorithms are found in silver. Greedy algorithms are local optimizers. Essentially, if a problem has a greedy solution, the best decision at any step is the global best decision. We can say that greedy problems a subset of or easier than dynamic programming problems.

   (a) **Non-example** Think about coin-change. I want to minimize the number of coins needed for a certain value. What you do is you use as much high value coins until you can't and then you use smaller value ones. Choosing the best decision locally is also the best decision globally.

   Not true in most cases because you don't really explore other options. DP is known for optimally exploring all choices.

   (a) **Example** Common economics example, you are a fish farm and you wanted to maximize the total amount of fish over a season. You have the following behavior or dynamics:

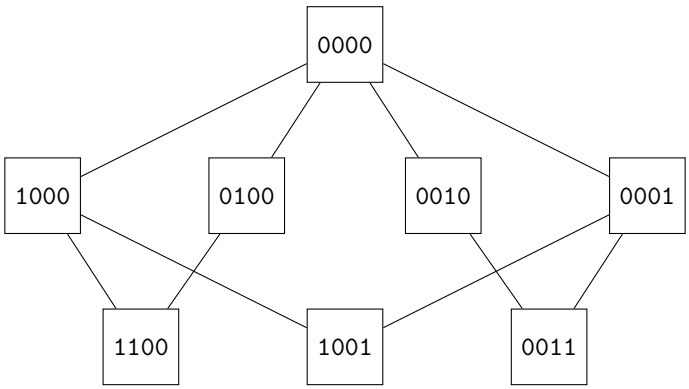
      i. Fishes in the fish tank will reproduce more fishes. Too little fishes, they lower yield. Too much fishes, they cluster and fight for resources reducing yield. There is a goldilock amount that will maximize yield.
      ii. Some other small nuances of production function.
      iii. You can harvest any amount of fishes at anytime during the season.. But after the season is over, you must harvest all of them.

      We can obviously see the greedy approach of harvesting all the fishes will result in a lower payout. We also know that the approach of never harvesting them until the very end is not a good approach either. The best approach is to harvest them near the goldilocks amount. Its not a greedy approach.

3. **Memorization** When you explore options, sometimes two choices derive their value from a common point. Think about initialization. All choices branch from the start position.

1.2 Bitmask DP

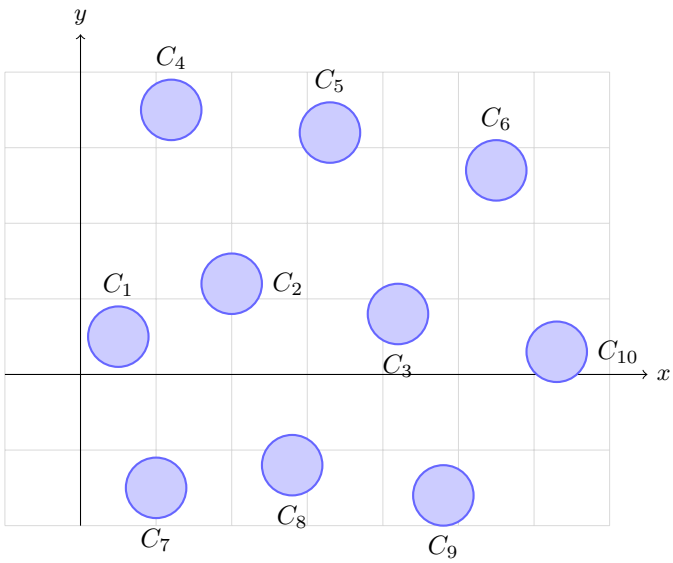
Application of DP. Bitmask dp is essentially iterating over all the possible configurations but only iterating over them once. You can view a string of "000(...)000" implying a initial configuration of nothing. And "000(...)001" then "000(...)010" as transition from the initial to include the first value and second value.



All possible states in the space must be set of traversals from the initial state to the intermediate state. Take 1001 for example, we can see that 1001 can be derived from two intermediate states 1000 and 0001. If using 1000 $\rightarrow$ 1001 is more optimal than 0001 $\rightarrow$ 1001, then we would set the value of 1001 to be that and vice versa. You are exploring all possible combinations, but you are choosing not to do recalculations.

1.3 Traveling sales person

You can solve the traveling sales person with bitmask dp. Traditionally, it would take factorial time because you are ordering the path you would take to visit all the cities. Lets say there are 10 cities to visit, you want to visit them in some order. It would take 10! calculations to find the minimum.



Possible routes include

1. $C_1 \rightarrow C_2 \rightarrow C_3 ... \rightarrow C_9 \rightarrow C_8 \rightarrow C_{10}$
2. $C_1 \rightarrow C_2 \rightarrow C_3 ... \rightarrow C_8 \rightarrow C_9 \rightarrow C_{10}$
3. ...
4. $C_{10} \rightarrow C_9 \rightarrow C_8 ... \rightarrow C_3 \rightarrow C_2 \rightarrow C_1$

We can simplify the problem down into a problem of bitmask dp asking whether or not we included a set of cities. So essentially we can say the best path in a TSP is given the paths on nine of the ten cities, which tenth city should we add.

1. We visit all the cities from 1 to 9 or the bitmask of $p_{10} = 1111111110$

2. We visit all the cities from 1 to 10 but not 9, or the bitmask of $p_9 = 1111111101$

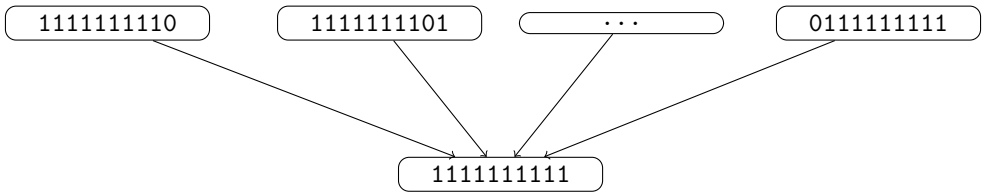
3. ...

4. We visit all the cities from 2 to 10, or the bitmask of $p_1 = 0111111111$

Let us name the set $S_{|path|=9}$. Then the answer or the minimum is just

$$min_{p_i \in S_{|path|=9}} DP[p_i] + min_{k \neq i} \text{distance } p \text{ to } i$$

The time complexity for this solution is $O(n2^n)$.



The 1111111111 node picks the most optimal choice.

1.4 Notes about time complexity

Bitmask DPs usually run in exponential time $O(2^n)$. You have to go through all combinations of flipping bits on n-variables. We often see it go from factorial time to exponential time. Decent amount of savings.

1.5 Notes for USACO

A very important skill in USACO is being able to read the constraints to a problem really well. Probably one of the easier DP problems if you can notice that the constraints to the problem are not that big. Bit DP problems usually require $O(2^n)$ because you are iterating over all the numbers, so if n is small, you probably have enough time to do it. Often times shrinks from factorial time to exponential. **HOWEVER** This DP optimizes these problems but do not make them "fast". TSP is a NP-hard problem, it cannot be solved in polynomial time. If you can map the problem to a NP problem and realize that the constraints are small, then you can detect what algorithm can be used.

2 Guard Mark

`https://usaco.org/index.php?page=viewproblem2&cpid=494` Pretty easy problem but a lot of bit manipulation needed. Brute force is factorial(20! worst case), but using DP is only (2^{20}).

2.1 Set up:

Define the types using structs. Easier to program with.

```
struct Cow {
    int height;
    int weight;
    int strength;
};
```

2.2 DP transition:

1. Initial Value: For DP problems, you have to pick values that are initialized. The zero state is when no cows are included. The max safety is infinity. We can use

   ```
   dp[0] = INT_MAX
   ```

2. Transition: Given a set of included cows from the bit representation of i, find the most safety you can get. For example:

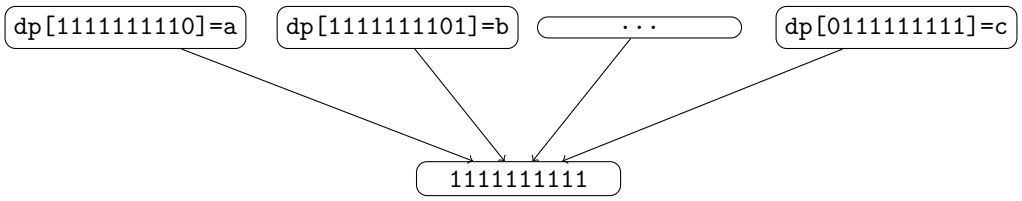
   (a) Initialize everything to the minimum.

   (b) Go through all possible inclusion and exclusion of binary strings.

   (c) Calculate the weight of the binary string. Recall that if you have any amount of cows above you, it doesn't matter which order they are in.

   (d) Choose the most optimal choice such that the stack maximizes the safety.

   (e) Find the safest subject to the constraint.



The DP[1111111111] will be the most optimal which is defined by min(dp[],strength - weight).

3 Moovie Mooving

`https://usaco.org/index.php?page=viewproblem2&cpid=515` Also pretty easy.

Note: Github link to solution: `TODO`

4 Recent 2023: Lights Off

`https://usaco.org/index.php?page=viewproblem2&cpid=1282`

4.1 Rotation of bits

One part of the problem that needs to be implemented is the shifting of bits with wrapping. Attached below is template for you to write the function. So for example the following should be the output:

1. 00110 → 00011
2. 00001 → 10000

```
#include <iostream>
#include <bitset>
#include <cstdlib>
#include <ctime>

int rotateRight_stringOrder(int mask, int N) {
    //write the code here
}
```

```
// Helper to print mask as N-bit binary
void printBin(int mask, int N) {
    std::bitset<32> b(mask);
    // Print only last N bits
    for (int i = N - 1; i >= 0; --i) {
        std::cout << b[i];
    }
}

int main() {
    //this will let you test on it
    std::srand(std::time(nullptr));
    int N = 8;  // Number of bits

    int mask = std::rand() % (1 << N); //random number

    std::cout << "Before rotation: ";
    printBin(mask, N);
    std::cout << " (" << mask << " decimal)\n";

    int rotated = rotateRight_stringOrder(mask, N);

    std::cout << "After rotation : ";
    printBin(rotated, N);
    std::cout << " (" << rotated << " decimal)\n";

    return 0;
}
```

You will also need to write the backwards version of this where you shift left and wrap to the 1 bit.

4.2 Equivalence Classes

In algebra, there is this concept of equivalence classes. Essentially given a set of objects, we can say objects in the set are equivalent to one another if a specific "relation" between the two make them similar.

Note: More specifically the following conditions must hold for any $a, b, c \in S$:

1. **Reflexivity**: $a \sim a$ is true. $\sim$ means relationship.
2. **Symmetry**: $a \sim b \rightarrow b \sim a$. if a is similar to b, then b must be similar to a.
3. **Transitivity**: $a \sim b, b \sim c \rightarrow a \sim c$. if a is sim to b and b is similar to c, then a must be similar to c.

For example, all the integers mod 3 with the same modulus are in the equivalence class.

1. $[0] = \{... - 3, 0, 3, 6, 9, ...\} \subset Z$
2. $[1] = \{... - 2, 1, 4, 7, ...\} \subset Z$
3. $[2] = \{... - 1, 2, 5, 8, ...\} \subset Z$

For all the sets, each element has the same modulus.

4.2.1 Rotating switches

In the problem, the third action is

> *Cyclically rotate the switches to the right by one. Specifically, if the bit string corresponding to the switches is initially $s_0 s_1 ... s_{N-1}$ then it becomes $s_{N-1} s_0 ... s_{N-2}$.*

This form the following equivalence class on three bits with the relation of rotating right by 1:

1. **Zero**: $\{000\}$
2. **Ones**: $\{100, 010, 001\}$
3. **Twos consecutive**: $\{110, 011, 101\}$
4. **threes**: $\{111\}$

The sum of all the equivalence class must be the original class size.

Code needed: The code for defining what equivalence class a binary number is in.

```
//code here
```

4.3 Left shift wrap is equivalent to right-shift wrap

We can prove that the equivalence class generated by the right-shift wrap is equivalent to the class generated by the left-shift wrap. Please refer to the blue note above for the three conditions of an equivalence class.

Proof

1. Let us have the equivalence class C of an arbitrary set on some sized-n binary numbers where $C = \{b_0, b_1, ...b_m\}$
2. From the third rule we know the following is true: $b_0 \sim b_1, b_1 \sim b_2, ...b_{m-1} \sim b_m$ then $b_0 \sim b_m$ by the third rule transitivity. **That is to say we can get b_m from apply the right-shift wrap after m-step from b_0.**
3. $b_0 \sim b_m \rightarrow b_m \sim b_0$ by the second rule. This is equivalent to the left-shift wraping on b_m once to get b_0. The inverse relation holds. We can apply this to get $b_m \sim b_{m-1} ... \sim b_0$ on the new relation.
4. The equivlance class on the two actions are the same then. Or that $b_i \sim b_{i-1}$ holds too with left-shift wrap.

Note: This holds because the relation is bijective or 1-1. Every input is mapped to only **one** output.

Not very important, but just another note. We can the set of natural numbers $N = \{0, 1, 2, 3, 4...\}$ with the relation ship $n + 1$. The natural set is the equivalence class itself. But there exists not inverse. Why? not bijective. There is no relation that maps to 0. The only way for this to be possible is if $-1 \in N$. But it is not. This would be invertible if we used all integers not just postitive + 0.

4.4 State Reduction

Given this set of switches and lights, should they have the same amount of steps?

1. Light: 011000, switch: 010101
2. Light: 001100, switch: 101010

Yes because they are simply right shifted both. This is where we would use the rotating switches classes.

5 Going from the very end

In another obscure version of dynamic programming is **calcualtion of variations**. The main idea behind this type of problem is an optimization problem. But instead you start from the very end and then work you way backwards. In our problem, we have the same principal. We start from the very end because we know that for all problems, the ending light configuration is off. That is the big idea behind this problem. So we can assume we start with the zero light and see if we can get to the light configuration of the given problem's input.

5.1 DP table

The build up to the DP table is a little harder to explain. But the idea here is we have a growing bitmask. For example, on three bits we have 001, 011, 111, 110, 100, 000, 001, 011, 111. We will run through this configuration.

5.2 Generating the DP table on three lines

1. Step 1 with 001. The subscripts means which group it is. We can see only 000 is

	000_0	001_1	010_1	011_3	100_1	101_3	110_3	111_7
0	1							
1								

   true then we check toggle 000 with 001. It is 001. Hence we set it as true. **Sanity**

	000_0	001_1	010_1	011_3	100_1	101_3	110_3	111_7
0	1							
1		1						

   Check: What we can see here is that 001 and all other elements in the set can be toggle on with only one move.

2. Step 2 with 011.

	000_0	001_1	010_1	011_3	100_1	101_3	110_3	111_7
1		1						
2								

   So the only true thing here is the 001, we then check **ALL ELEMENTS IN THE CLASS G_1 or 001,010,100**.

   (a) $011 \oplus 001 = 010 \in G_1$

   (b) $011 \oplus 010 = 001 \in G_1$

   (c) $011 \oplus 100 = 111 \in G_7$

So what we see here is that it takes at least two steps to do anything in group G_1(takes less) or the $G_7 = \{111\}$.

Note: Double check: How would you turn off the 111 light given that you start out with switch 000

.

	000_0	001_1	010_1	011_3	100_1	101_3	110_3	111_7
1		1						
2		1						1

3. Step 3 with 111.

	000_0	001_1	010_1	011_3	100_1	101_3	110_3	111_7
2		1						1
3								

We do the same process and check it on four elements now:

(a) $111 \oplus 001 = 110 \in G_2$

(b) $111 \oplus 010 = 101 \in G_2$

(c) $111 \oplus 100 = 011 \in G_2$

(d) $111 \oplus 111 = 000 \in G_0$

And this means that we can do anything in the second class G_2 with three steps.

Note: Double check: How would you turn off the 101 light given that you start out with switch 000 Does it really take three steps.

.

	000_0	001_1	010_1	011_3	100_1	101_3	110_3	111_7
2		1						1
3	1		1					

We have technically done every class already but we have to repeat the process for the other few steps. After doing so you would have generated all the neccessary steps to calculate how much steps it takes to turn off all the lights given a starting switch.

6 Final steps

Now all we have to do is calculate the number of step it will take if we are given a starting light configuration and a switch configuration. To do this, what we need to do is track only the rotation of switches at each step. We can do this with a loop easily

1. initialize current = 0
2. For all steps $1 \rightarrow 3 * N$:
   (a) current = current $\oplus$ switch
   (b) rotate the switch left

However this doesn't take into account the toggles + rotates. Luckily, the DP table takes into account of toggles + rotates but not the initial starting switch configuration. So all we have to do is XOR with the starting configuration. The following (current $\oplus$ light) actually gives us the light configuration if we were to start from the zero light and the

zero switch. We know that anything xored with zero is itself so (light $\oplus$ 000...000) = light and this is also applied to the switch.

So essentially the idea is the DP keeps track of the states starting at no lights and no switches but we add in a switch every time and rotate. But if we XOR with the starting light configuration and also calculate what the starting configuration of the switches will be at the i-th step we can effectively use the DP table.