Bitmask DP

UF Programming Team

What is a bit?

A ***bit*** is the basic unit of information in computing

A bit can only have two values: ***0*** or ***1***

These two values can also be called ***true*** and ***false***, “***on***” and “***off***”, etc.

Numbers can be represented using bits: ***base-2 (or binary)***

$\rightarrow 16_{10} == 10000_2$

$\rightarrow 54_{10} == 110110_2$

But how are these numbers equal?

Binary Numbers

To determine what ***base-10*** number a binary numbers represents, you can use the following algorithm:

1) List out the powers of 2 above each bit, going right to left
2) For each bit in the binary number, if the value of the bit is a 1, add its corresponding power of 2 to a running sum
3) After going through all bits, the running sum will be the ***base-10*** representation of the number

Binary to Decimal Example

Convert ***1001101*** $_{\boldsymbol{2}}$ to a base-10 number

64	*32*	*16*	*8*	*4*	*2*	*1*
1	**0**	**0**	**1**	**1**	**0**	**1**

64 + 0 + 0 + 8 + 4 + 0 + 1 = **77**

$\rightarrow$ ***1001101*** $_{\boldsymbol{2}}$ **== *77*** $_{\boldsymbol{10}}$

Bitwise Operations

Now that we know how to represent numbers as binary strings, we will take a look at some operations we can perform

→ **NOT**

→ **AND**

→ **OR**

→ **Bit shifts**

NOT

The bitwise **NOT**, or complement, is an operation that performs *negation*, which means bits that are 0 become 1, and those that are 1 become 0

1011001 → ***0100110***

111111 → ***000000***

100000 → ***011111***

In Java, the bitwise **NOT** is represented by **~**

→ ~32 == 31

AND

The bitwise **AND** is an operation between two binary numbers that performs the *logical AND* operation on each pair of the corresponding bits, which means that if both bits are 1, the resulting bit is 1, and any other combination results in a 0 bit

→ **TRUE && TRUE == TRUE**

→ **TRUE && FALSE == FALSE**

→ **FALSE && FALSE == FALSE**

In Java, the bitwise **AND** is represented by **&**

→ 35 & 31 == 11

AND

Let’s look at an example

```
      1100110
AND   0110101
-----------------------
      0100100
```

OR

The bitwise **AND** is an operation between two binary numbers that performs the *logical OR* operation on each pair of corresponding bits, which means that if both bits are 0, the resulting bit is 0, and any other combination results in a 1 bit

→ **TRUE || TRUE == TRUE**

→ **TRUE || FALSE == TRUE**

→ **FALSE || FALSE == FALSE**

In Java, the bitwise **OR** is represented by **|**

→ 35 | 31 = 63

OR

Let’s look at an example

```
        1100110

OR     0110101
------------------------
       1110111
```

Bit shifts

The ***bit shift*** operation moves (or shifts) digits to the left or right in a binary number

In Java, a ***left-shift*** is denoted by **<<**, and a ***right-shift*** is denoted by **>>**

You will also need to declare how many bits you want to shift a number

→ $\boldsymbol{101_2 \ll 2 == 10100_2}$

→ $\boldsymbol{100101_2 \gg 3 == 100_2}$

→ $\boldsymbol{1_2 \ll 5 == 100000_2}$

Bit shifts

Let's look at the following pattern of bit shifts

$1 << 0 == 1_2 == 1_{10}$

$1 << 1 == 10_2 == 2_{10}$

$1 << 2 == 100_2 == 4_{10}$

$1 << 3 == 1000_2 == 8_{10}$

What's the pattern?

→ ***$1 << k == 2^k$*** (*we will be using this later!*)

Subsets

Let ***A*** = {2, 3, 5, 7}, let ***B*** be a subset of ***A***

B is a subset of ***A*** if all elements in ***B*** are also elements in ***A***

$\longrightarrow$ ***B*** = {2, 3, 5}

$\longrightarrow$ ***B*** = {5, 7}

$\longrightarrow$ ***B*** = {2, 3, 5, 7}

$\longrightarrow$ ***B*** = { }

Subsets

So how can we determine how many subsets there are of a set, ***S***?

We can say that for each element in ***S***, we have two choices:

→ put the element in our subset

→ don’t put the element in our subset

Because we have two choices per element, the total number of subsets we can have can be calculated with 2^n, where ***n*** is the number of elements in ***S***

Subsets

S = {2, 3, 5, 8}

Subsets of S:

0-element: { }

1-element: {2}, {3}, {5}, {8}

2-element: {2, 3}, {2, 5}, {2, 8}, {3, 5}, {3, 8}, {5, 8}

3-element: {2, 3, 5}, {2, 3, 8}, {2, 5, 8}, {3, 5, 8}

4-element: {2, 3, 5, 8}

Subsets

In order to create a subset, we either *take* or *ignore* an element in the set

So what can we use to model these subsets? (*think binary!*)

Let’s create a binary string of length ***n*** (for each element in our set)

$\rightarrow$ For a bit some index ***k*** in our binary string

- $\Rightarrow$ if the bit at ***k*** is 0, then our subset ***does not*** contain the element at ***k***
- $\Rightarrow$ if the bit at ***k*** is 1, then our subset ***does*** contain the element at ***k***

Subsets

S = {2, 3, 5, 8}

There are four elements in this set, so we will need to create a binary string where ***n* = 4**

0	***1***	***2***	***3***
2	3	5	8

Let’s say that our subset is {3, 8}. What will the corresponding binary string look like? (*remember that indices in a binary string are read from right to left!)*

$\rightarrow \boldsymbol{1010_2 == 10_{10}}$

Subsets

S = {2, 3, 5, 8}

What is the binary string representation of the following subsets?

{2, 3}

→ $\boldsymbol{0011_2 == 3_2}$

{2, 3, 5, 8}

→ $\boldsymbol{1111_2 == 15_{10}}$

{ }

→ $\boldsymbol{0000_2 == 0_{10}}$

Subsets

So how can we easily iterate over ***all*** subsets of a set, ***S***?

We know that an *empty* subset has a binary string of $\boldsymbol{0_2}$, which is equal to ***0***

We also know that a *full* subset has a binary string of $\boldsymbol{1111...1111_2}$, which is equal to $\boldsymbol{2^n - 1}$

Now that we know all subsets fall in the range $\boldsymbol{[\ 0,\ 2^n - 1\ ]}$, we can iterate over all numbers in this range and get all subsets of our set ***S***

Subsets

Recall that ***1 << k*** == ***2***^{k}

```
for ( int subset = 0; subset < (1 << n); subset++ ) {
    // This for-loop will find every subset of a set
}
```

Now that we have our for-loop set up, how do we use the current value of **`subset`** to determine which elements are in our subset?

→ We just check which bits in **`subset`** are 1

Subsets

Given a binary string, how can we check if the bit at index ***k*** is 0 or 1?

7	***6***	***5***	***4***	***3***	***2***	***1***	***0***
1	0	0	1	0	1	1	1

Let’s say we want to see if the bit at ***k* = *3*** is 0 or 1, what can we do to figure this out?

→ **AND?**

→ **OR?**

→ **Bit shifting?**

Subsets

For starters, let’s using left-shifting to focus in on the bit at ***k* = 3**

We will create a new binary string whose value is ***1 << 3***

7	***6***	***5***	***4***	***3***	***2***	***1***	***0***
1	0	0	1	0	1	1	1

0	0	0	0	1	0	0	0

What can we do now to check to see if the bit at ***k* = 3** is a 0 or 1?

→ We will use **AND**!

Subsets

7	***6***	***5***	***4***	***3***	***2***	***1***	***0***
1	0	0	1	0	1	1	1

0	0	0	0	1	0	0	0

==================================

0	0	0	0	0	0	0	0

When we **AND** our original binary string with ***1 << 3***, we get ***0***

Subsets

Now let's try and find the value of the bit at ***k* = 2**

7	***6***	***5***	***4***	***3***	***2***	***1***	***0***
1	0	0	1	0	1	1	1

0	0	0	0	0	1	0	0

=====================================

0	0	0	0	0	1	0	0

When we **AND** our original binary string with ***1 << 2***, we get ***4***

Subsets

Given a binary string, how can we use left-shift and the bitwise operation **AND** to determine if the bit at index ***k*** is 0 or 1?

```
int leftShift = 1 << k;
int result = subset & leftShift;
if ( result == 0 )
    // the bit at index k is 0!
else
    // the bit at index k is 1!
```

Bitmask DP

Now that we know how to represent subsets as binary strings, we can now take a look at ***bitmask dynamic programming***

Bitmask DP is a type of dynamic programming that uses *bitmasks* in order to keep track of our current state in the problem

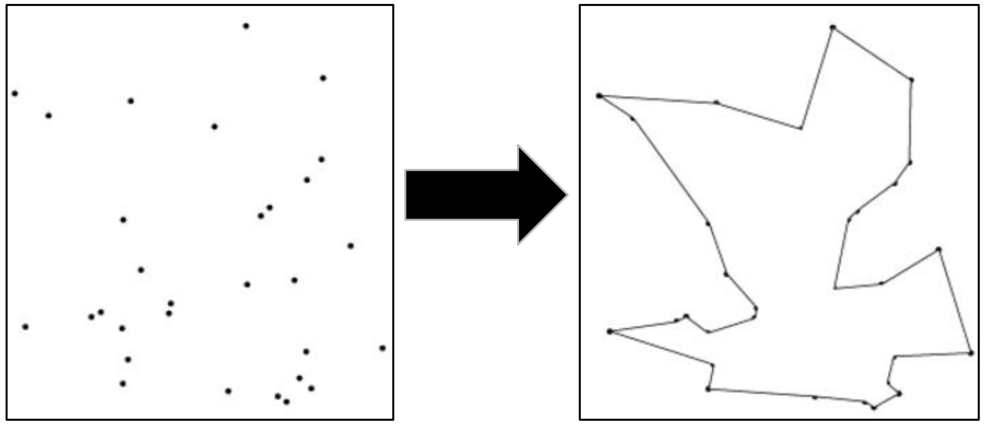
A bitmask is nothing more than a number that defines which bits are on and off (*binary string representation of the number*)

Let’s look at a classic example in order to see how a bitmask can be used in dynamic programming

Traveling Salesman Problem

Condensed Problem Statement

Given a list of cities, as well as the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns back to the city that you started from?



Traveling Salesman Problem

Imagine that we have ***N*** cities that we need to travel to

We can create a permutation of the ***N*** cities to tell us what order we should visit the cities

→ If we have ***N = 4***, we could have {3, 1, 4, 2} which tells us the order to visit the cities in: 3 → 1 → 4 → 2 → 3 (*we return home at the end*)

How many permutations could be possibly have with ***N*** cities?

→ ***N!*** *(**N** choices for the first city, **N-1** choices for the next city, **N-2** for the next, etc.)*

But if ***N*** > 10, this algorithm will be too slow; let’s shoot for something better

Traveling Salesman Problem

Let’s use bitmasks to help us with this problem; we’ll construct a bitmask of length ***N***. What does the bit at index ***k*** represent in the scheme of the problem? What if it’s 0? What if it’s 1?

→ If the bit at ***k*** is 0, we ***have not yet*** visited city ***k***

→ If the bit at ***k*** is 1, we ***have*** visited city ***k***

For a given bitmask, we can see which cities we have left to travel to by checking which bits have a value of 0

Traveling Salesman Problem

This bitmask will work, but is it the only piece of information we need?

Let’s say we have the following bitmask ***10011***

This bitmask tells us we have yet to visit cities ***2*** and ***3***

However, if we try and visit these cities, the bitmask ***does not*** tell us the current city we are at, so there’s no way for us to know the distance it takes to go from our current location (with our current bitmask) to city 2 or city 3

In addition to the bitmask, we need to keep track of our ***current position***!

Traveling Salesman Problem

Now that we keep track of our current position and a bitmask that tells us which cities we have left to visit, we can construct a recursive function that will give us the answer we desire

```
public static int solve( int bitmask, int position ) {

    // return the minimum distance to visit all unvisited

    // cities in the bitmask and go back to the origin from

    // the city we are currently at

}
```

Traveling Salesman Problem

If we know **`solve( bitmask, position )`** will give us the answer to the Traveling Salesman Problem, and we say that we start at city ***0***, we should we replace **`bitmask`** and **`position`** with so that we get our answer?

→ **`solve( 1, 0 )`**

⇒ our starting bitmask is 000...0001 (since we have visited city ***0***)

⇒ our current position is city ***0***

Traveling Salesman Problem

Can we come up with a base case(s) for this function?

→ What if we have visited ***all*** of the cities? What does this bitmask look like?

⇒ ***1111…1111*** == ***2^N - 1***

So if ***bitmask == 2^N - 1***, what do we return?

→ return the distance from your current position to the origin city

Traveling Salesman Problem

So what do we do when we still have cities left to visit?

```
int minDistance = Integer.MAX_VALUE;

for ( int k = 0; k < N; k++ ) {

    if ( (bitmask & ( 1 << k )) == 0 ) {

        int newDistance = ???

        minDistance = Math.min(minDistance, newDistance);

    }

}
```

Traveling Salesman Problem

What is our **`newDistance`**?

We need to figure out the minimum distance we get from traveling to city ***k*** and then traveling to unvisited cities and back to the origin

→ This is just **`dist[position][k] + solve( bitmask, k )`**

But since we visited city ***k***, we need to update the bitmask

How do we switch the bit at index ***k*** from 0 to 1?

→ **`bitmask = bitmask | (1 << k);`**

Traveling Salesman Problem

```
int minDistance = Integer.MAX_VALUE;

for ( int k = 0; k < N; k++ ) {

    if ( (bitmask & ( 1 << k )) == 0 ) {

        int newDistance = solve( bitmask | ( 1 << k ), k ) + dist[position][k];

        minDistance = Math.min( minDistance, newDistance );

    }

}

return minDistance;
```

Problems

Split up into groups of 2-3 and work on the following problem

goo.gl/LHsqCc

Other problems

** goo.gl/AYHO76

** codeforces.com/gym/100712 *(problem G)*