DP II

Sep 19: Lesson 3

Attendance

ICPC Interest Form

https://tinyurl.com/iwanttogotoicpc

Recap

What is DP?

State - a set of unique subproblems defined by parameters

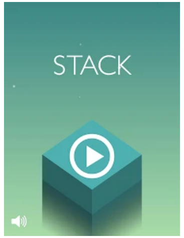
Transition - how to calculate subproblems from smaller cases.

Classic Algorithms

- Longest Increasing Subsequence
- Coin Change, Coin Combinations
- Longest Common Subsequence
- 0-1 Knapsack
- etc.

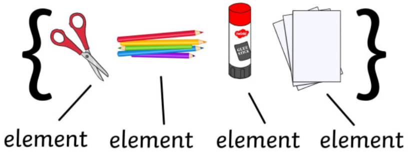
STL Speed Ups

STL Speed Ups



vroom vroom

vroom vroom
vroom vroom



STL Speed Ups

std::set

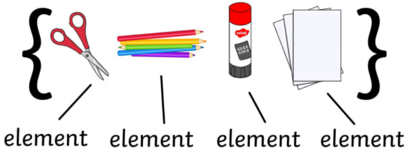
Defined in header <set>

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

(1)

```
namespace pmr {
    template<
        class Key,
        class Compare = std::less<Key>
    > using set = std::set<Key, Compare, std::pmr::polymorphic_allocator<Key>>;
}
```

(2) (since C++17)



STL Speed Ups

class template

<queue>

std::queue

```
template <class T, class Container = deque<T> > class queue;
```

FIFO queue

queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the ***"back"*** of the specific container and ***popped*** from its ***"front"***.

STL Speed Ups

class template

std::deque

Defined in header <deque>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```
(1)

```
namespace pmr {
    template< class T >
    using deque = std::deque<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

std::deque (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. In addition, insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements.

STL Speed Ups

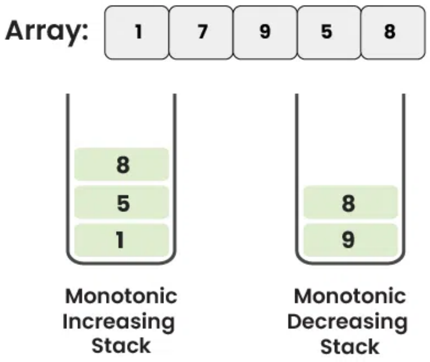
std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adaptor that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.



Easy Problem

let’s warm up :)

Candy Bandit

Problem Statement:

- There are **N** ≤ 500 000 tiles lined up from Raymond's house to CoC 052
- To step on tile **i,** Raymond must have give Marianna **A[i]** candies; if he has already given me **A[i]** candies or more, he can step on the tile for free, otherwise, he has to hand over more candies until I receive **A[i]** candies. Raymond can jump at most **K** ≤ 10 tiles forward at one time. It is assumed that Raymond can jump from the his home to the first K tiles and he can jump from any of the last K tiles to the classroom.
- Raymond wants his candies, so he wants to minimise the maximum **A[i]** of all the tiles he steps on.

Candy Bandit

Solution:

N = 10**, K** = 3

0 1 3 2 4 8 2 7 3 5

Candy Bandit

Solution:

N = 10**, K** = 3

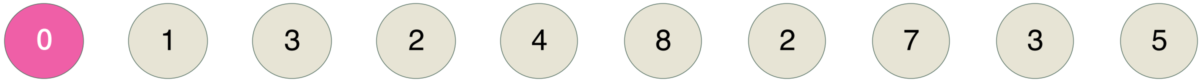
0	1	3	2	4	8	2	7	3	5

From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10, **K** = 3

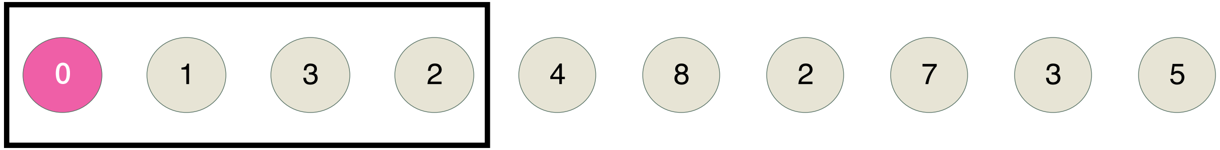


From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10, **K** = 3

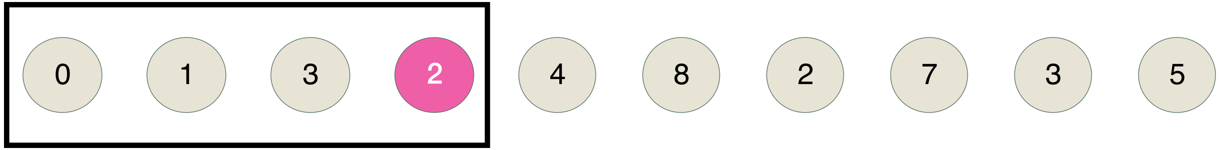


From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10, **K** = 3

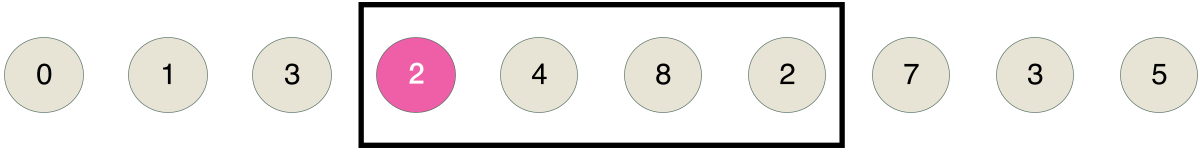


From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10, **K** = 3

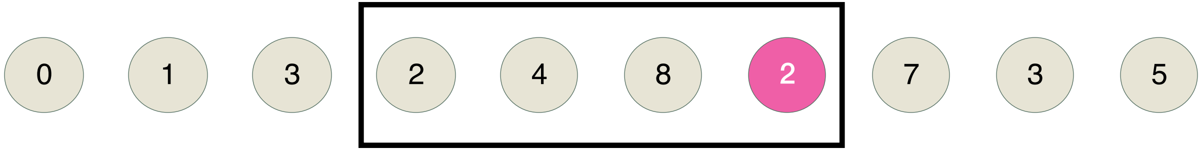


From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10, **K** = 3

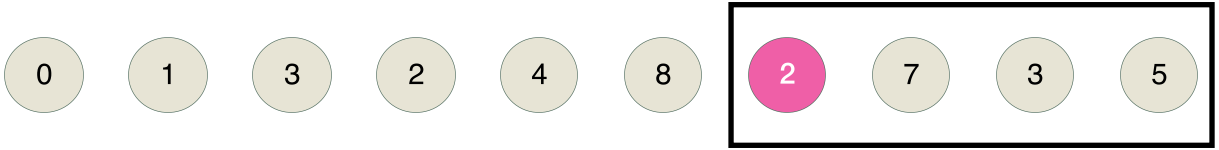


From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10, **K** = 3

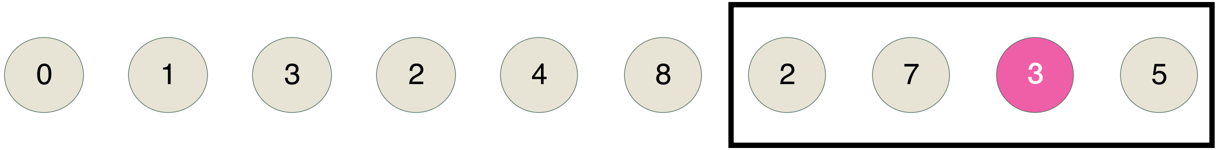


From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10, **K** = 3



From each tile i, Raymond could have come from any tile from i - K to i - 1

Candy Bandit

Solution:

N = 10**, K** = 3

- **State:** let dp(i) = max min tile required to reach tile i
- **Transition:** dp(i) = max(min(dp(i - K), …, dp(i - 1)), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

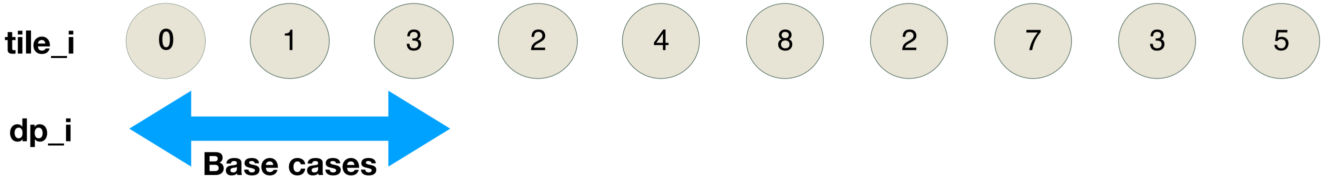
tile_i	0	1	3	2	4	8	2	7	3	5
dp_i										

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10, **K** = 3



$$dp(i) = \max(\min(dp(i - K), \ldots, dp(i - 1)), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**							

dp(i) = max(min(dp(i - K), …, dp(i - 1)), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**						

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**						
				2						

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2** **2** **3**						

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**						

$$dp(i) = \max(\min(dp(i - K), \ldots, dp(i - 1)), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**					

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**					
					4					

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**					
					4					
					4					

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**					

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**				

$$dp(i) = \max(\min(dp(i - K), \ldots, dp(i - 1), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**				
						8				

$$dp(i) = \max(\min(dp(i - K), \ldots, dp(i - 1), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**				
						8				
						8				

$$dp(i) = max(min(dp(i - K), \ldots, dp(i - 1), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**				

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**	**2**			

$$dp(i) = max(min(dp(i - K), \ldots, dp(i - 1), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**	**2** **4**			

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**	**2** **4** **8**			

$$dp(i) = \max(\min(dp(i - K), \ldots, dp(i - 1), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**	**2**			

$$dp(i) = \max(\min(dp(i - K), \ldots, dp(i - 1)), tile\_i)$$

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**	**2**	**7**	**3**	**5**

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

Candy Bandit

Solution:

N = 10**, K** = 3

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**	**2**	**7**	**3**	**5**

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

ans = dp(n)

Candy Bandit

Solution:

N = 10, **K** = 3

min

tile_i	0	1	3	2	4	8	2	7	3	5
dp_i	**0**	**1**	**3**	**2**	**4**	**8**	**2**	**7**	**3**	**5**

dp(i) = max(min(dp(i - K), …, dp(i - 1), tile_i)

ans = **dp(n)**

Let’s make this a little harder

still a warm up :)

Candy Bandit_ex

Problem Statement:

- What if **K** ≤ 10 is removed?
- Raymond could possibly jump **much further**
- dp(i) = max(dp**(i - K)...dp(i - 1)**, tile(i))
- The transition could now take up to **O(N)**

Candy Bandit_ex

- Notice how the “window” of max shifts forward by one position every iteration.
- We could use a sliding set on the DP values!
- Now the transition takes O(log N) , bringing the overall complexity down to O(N log N) .

Candy Bandit_ex

Solution:

N = 10, **K** = 3

0 1 3 2 4 8 2 7 3 5

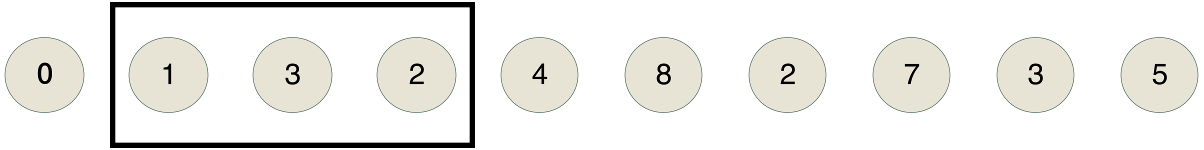
From each tile i, Raymond could have come from any tile from i - K to i - 1

{0, 1, 3}

Candy Bandit_ex

Solution:

N = 10**, K** = 3



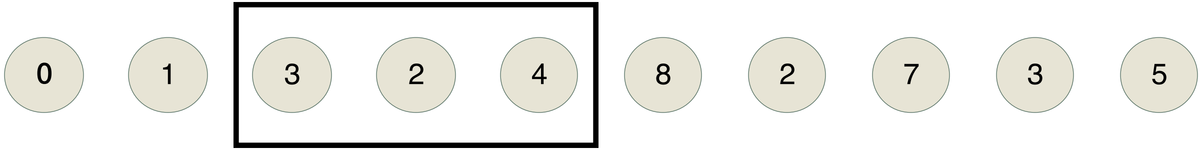
From each tile i, Raymond could have come from any tile from i - K to i - 1

{1, 3, 2}

Candy Bandit_ex

Solution:

N = 10**, K** = 3



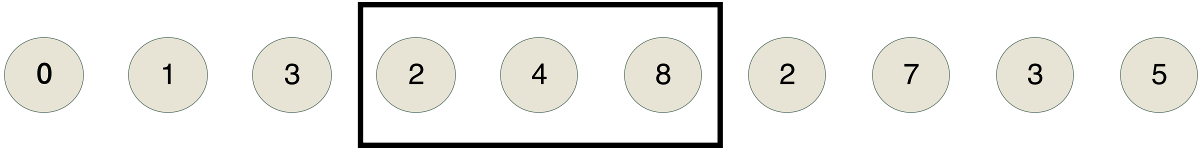
From each tile i, Raymond could have come from any tile from i - K to i - 1

{3, 2, 4}

Candy Bandit_ex

Solution:

N = 10**, K** = 3



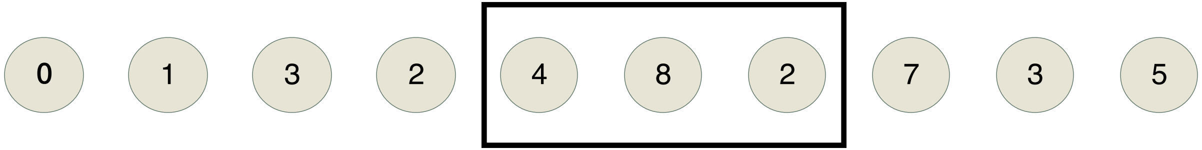
From each tile i, Raymond could have come from any tile from i - K to i - 1

{2, 4, 8}

Candy Bandit_ex

Solution:

N = 10, **K** = 3



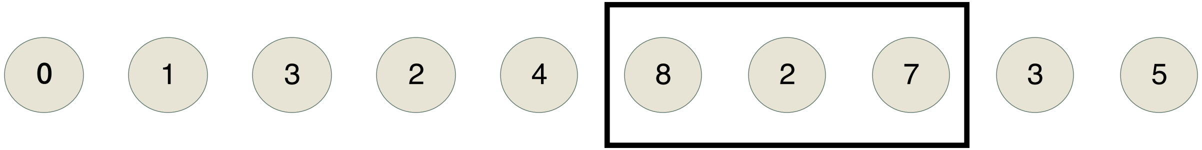
From each tile i, Raymond could have come from any tile from i - K to i - 1

{4, 8, 2}

Candy Bandit_ex

Solution:

N = 10**, K** = 3



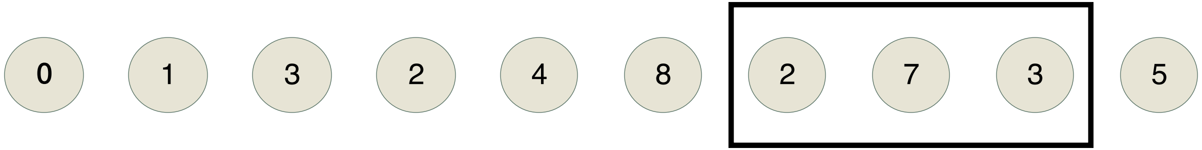
From each tile i, Raymond could have come from any tile from i - K to i - 1

{8, 2, 7}

Candy Bandit_ex

Solution:

N = 10**, K** = 3



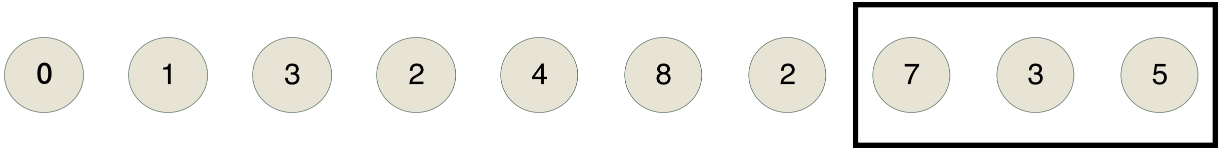
From each tile i, Raymond could have come from any tile from i - K to i - 1

{2, 7, 3}

Candy Bandit_ex

Solution:

N = 10**, K** = 3

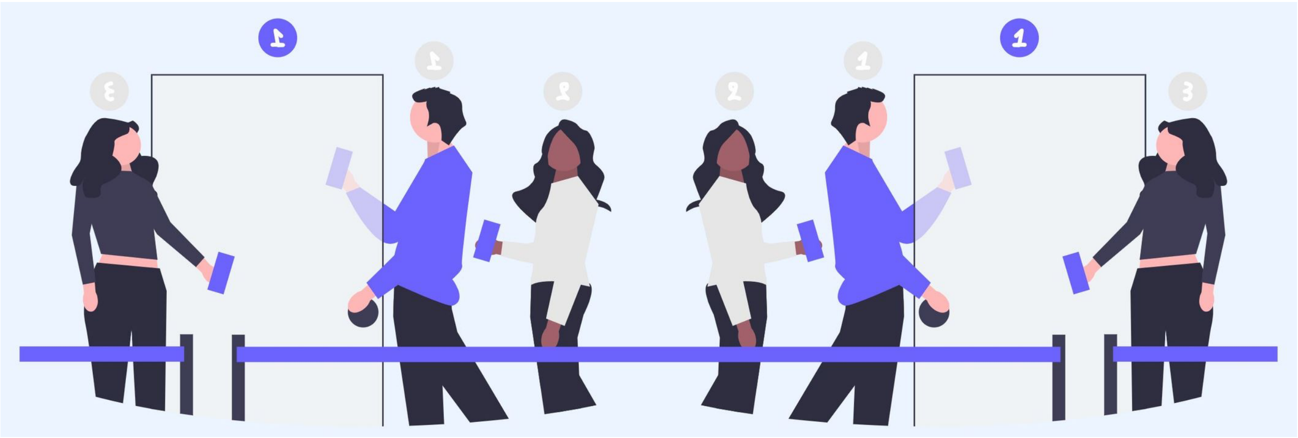


From each tile i, Raymond could have come from any tile from i - K to i - 1

{7, 3, 5}

Candy Bandit_ex

- Can you do it using a deque?



State Manipulation

State Manipulation

DP takes O(state + $\sum$ state transition) time.

State Manipulation

DP takes O(state + ∑ state transition) time.

You can exploit monotonicity to

- Reduce the state space
- Reduce the transition time

Watching (JOI 2013)

You want to take photos of $N \leq 2000$ events. The events are arranged on a line.

You have $P \leq 100{,}000$ small cameras and $Q \leq 100{,}000$ large cameras.

A small camera has a width of w, and a large camera has a width 2w.

The larger the parameter w is, the higher the cost to take pictures is.

Minimize the value of w.

https://oj.uz/problem/view/JOI13_watching

Watching (JOI 2013)

- $1 \leqq N \leqq 2\,000$.
- $1 \leqq P \leqq 100\,000$.
- $1 \leqq Q \leqq 100\,000$.
- $1 \leqq A_i \leqq 1\,000\,000\,000$ $(1 \leqq i \leqq N)$.

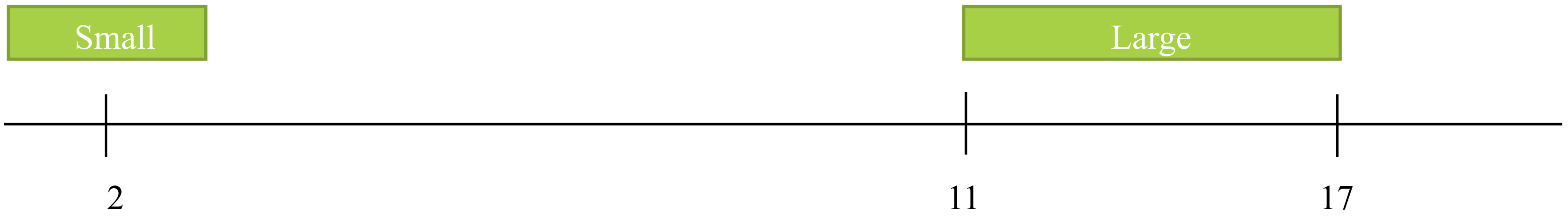
1000ms

256 MB

https://oj.uz/problem/view/JOI13_watching

Watching (JOI 2013)

w = 3



Watching (JOI 2013)

Observation 1

$w \geq 1$.

If we have $\geq N$ cameras, we can cover everything with $w = 1$.

Hence we can assume P, Q, $P+Q < N$

Watching (JOI 2013)

Observation 2

If w can cover everything, w+1 can also cover everything.

Find the minimum w → Binary search on w!

Watching (JOI 2013)

Observation 2

If w can cover everything, w+1 can also cover everything.

Find the minimum w → Binary search on w!

Watching (JOI 2013)

Reduced problem

Given a fixed w, can we cover all N points with P small and Q large cameras?

Watching (JOI 2013)

Instant DP

Sort the events!

Watching (JOI 2013)

Instant DP

Sort the events!

f(**i** = event i, **p** , **q**) =

f(**i** - **x** , **p** - 1, **q**) or f(**i** - **y** , **p** , **q** - 1)

where **x** , **y** = events covered by the small and large camera when the at right edge is at i.

Watching (JOI 2013)

Instant DP

Sort the events!

f(**i** = event i, **p** , **q**) =

f(**i** - **x** , **p** - 1, **q**) or f(**i** - **y** , **p** , **q** - 1)
where **x** , **y** = events covered by the small and large camera when the at right edge is at i.

State: O(N^3). Transition: O(1).

Watching (JOI 2013)

Instant DP

Sort the events!

$f(\mathbf{i} = \text{event } i, \mathbf{p}, \mathbf{q}) =$

$f(\mathbf{i} - \mathbf{x}, \mathbf{p} - 1, \mathbf{q})$ or $f(\mathbf{i} - \mathbf{y}, \mathbf{p}, \mathbf{q} - 1)$

where $\mathbf{x}$, $\mathbf{y}$ = events covered by the small and large camera when the at right edge is at i.

State: O(N^3). Transition: O(1).

Watching (JOI 2013)

Welp...

O(N^3) state

N ≤ 2000 – too big!

We need to reduce the state space.

Watching (JOI 2013)

Observation 3

The value of our DP is true/false.

Can we store more information within the value?

Notice if $f(i, p, q)$, then $f(i, p+1, q)$, and $f(i, p+2, q)$, and ...

Watching (JOI 2013)

Solution!

Remove p from the state!

f(i, q) = minimum p such that it is possible to cover everything.

State: O(N^2)
Transition: O(1)

Watching (JOI 2013)

Solution!

Remove p from the state!

$f(i, q) = \min(f(i - x, q) + 1, f(i - y, q - 1))$

State: O(N^2)
Transition: O(1)

Watching (JOI 2013)

Alternative Solution!

What about

$f(p, q)$ = maximum **i** that you can cover with **p** small and **q** large cameras?

Watching (JOI 2013)

Alternative Solution!

What about

$f(p, q) = \max(f(p-1, q) + x , f(p, q-1) + y)$

where **x** , **y** = events covered by the small and large camera.

Watching (JOI 2013)

Key idea

– DP takes O(state + ∑ state transition) time.

– You can exploit **monotonicity** to

- Reduce the state space
- Reduce the transition time

Digit DP

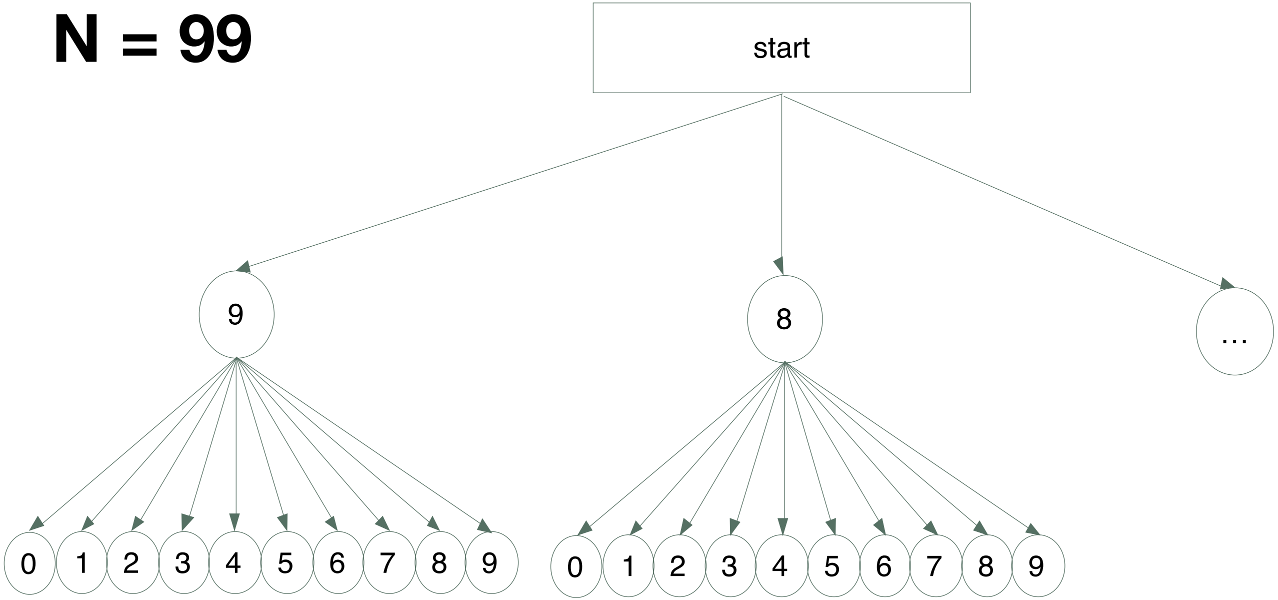
Does that mean everything before this was analog?

Digit DP

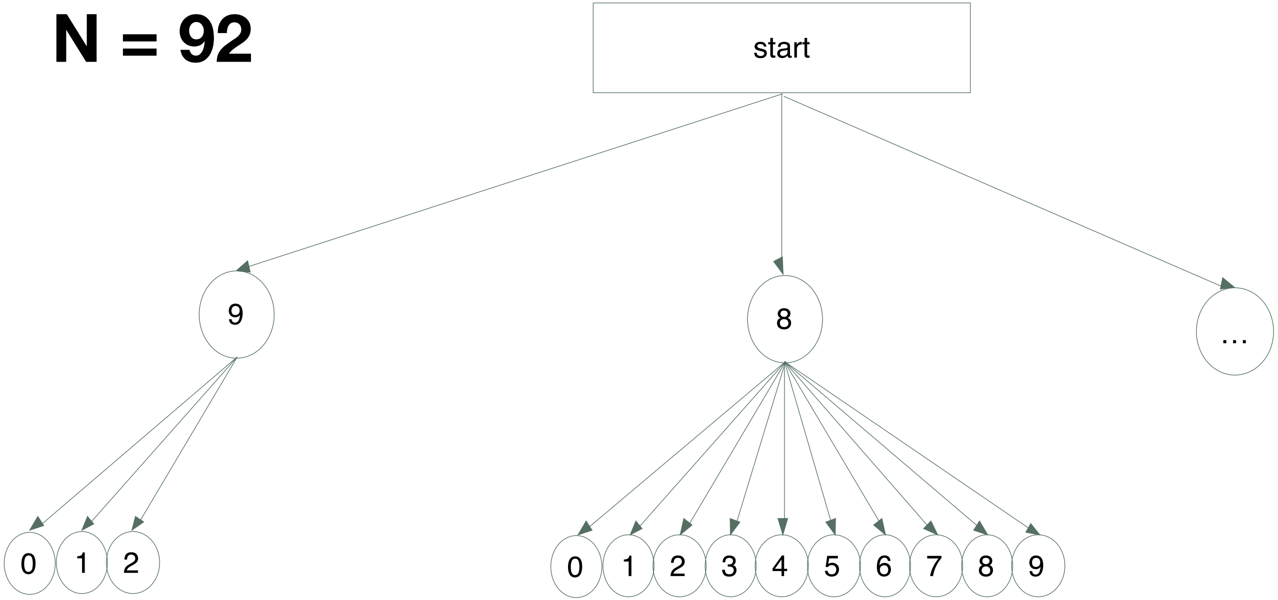
Find the number of integers ≤ N which contain no two consecutive equal digits. (ignore leading zeros)

N ≤ 10^18

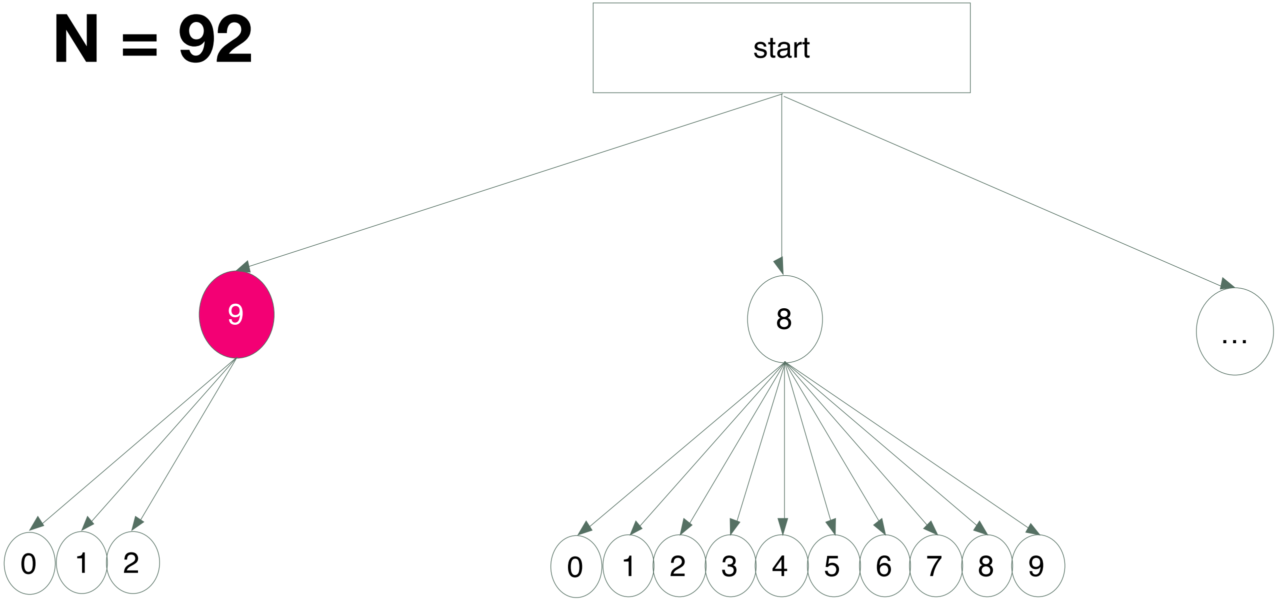
N = 99



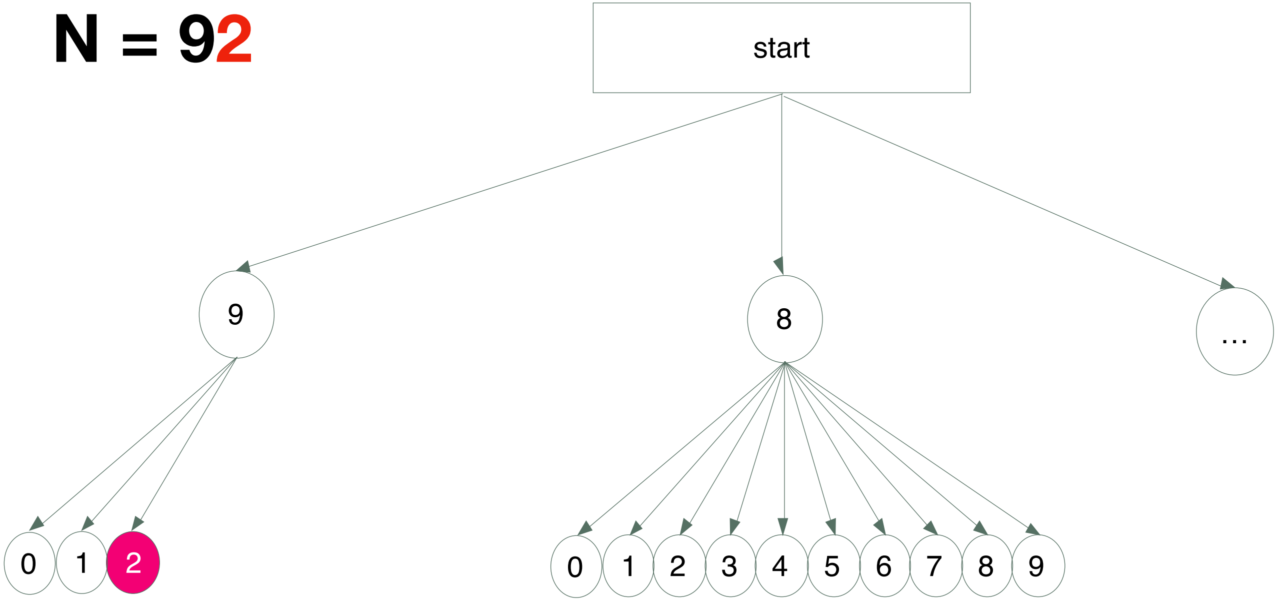
N = 92



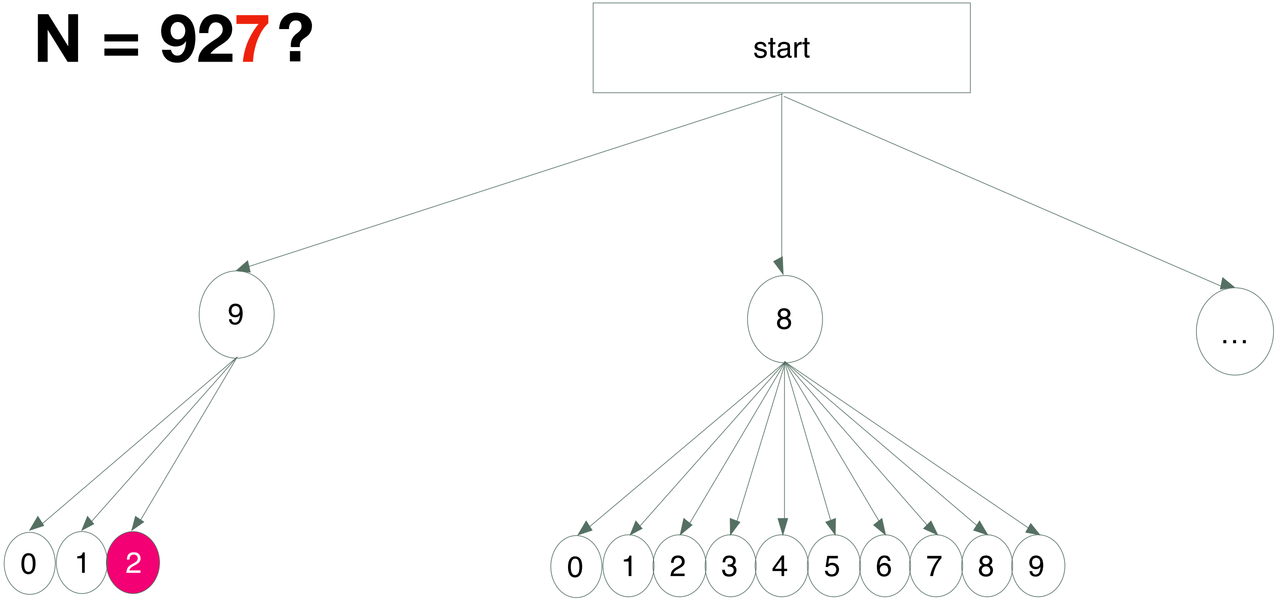
N = 92



N = 92

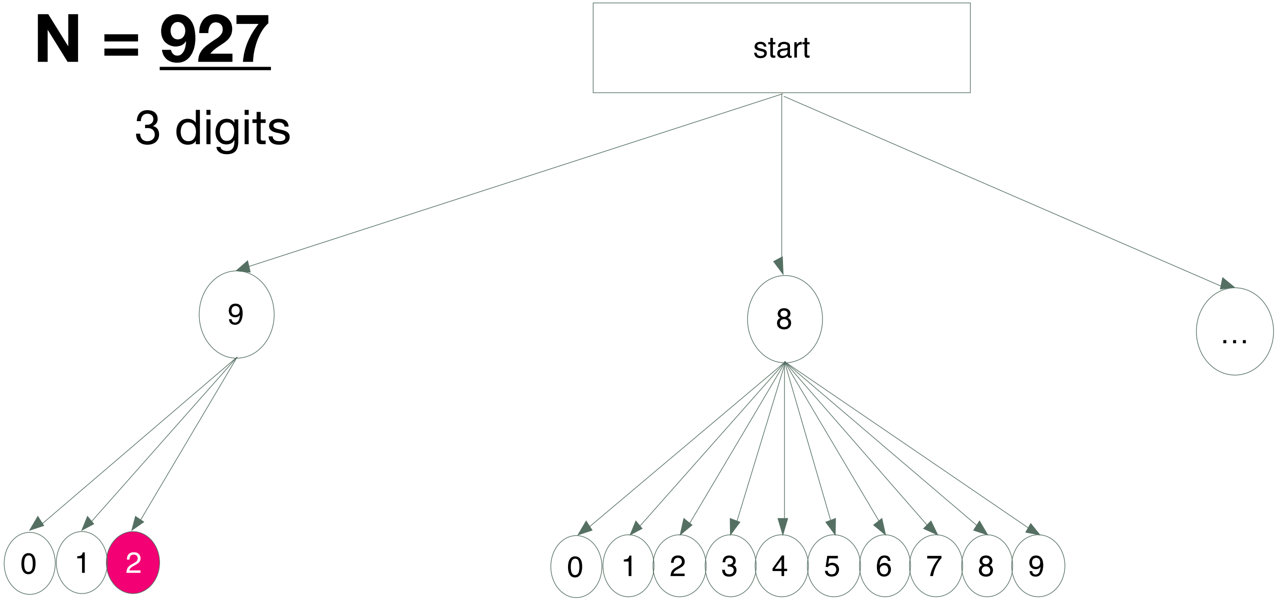


N = 927?



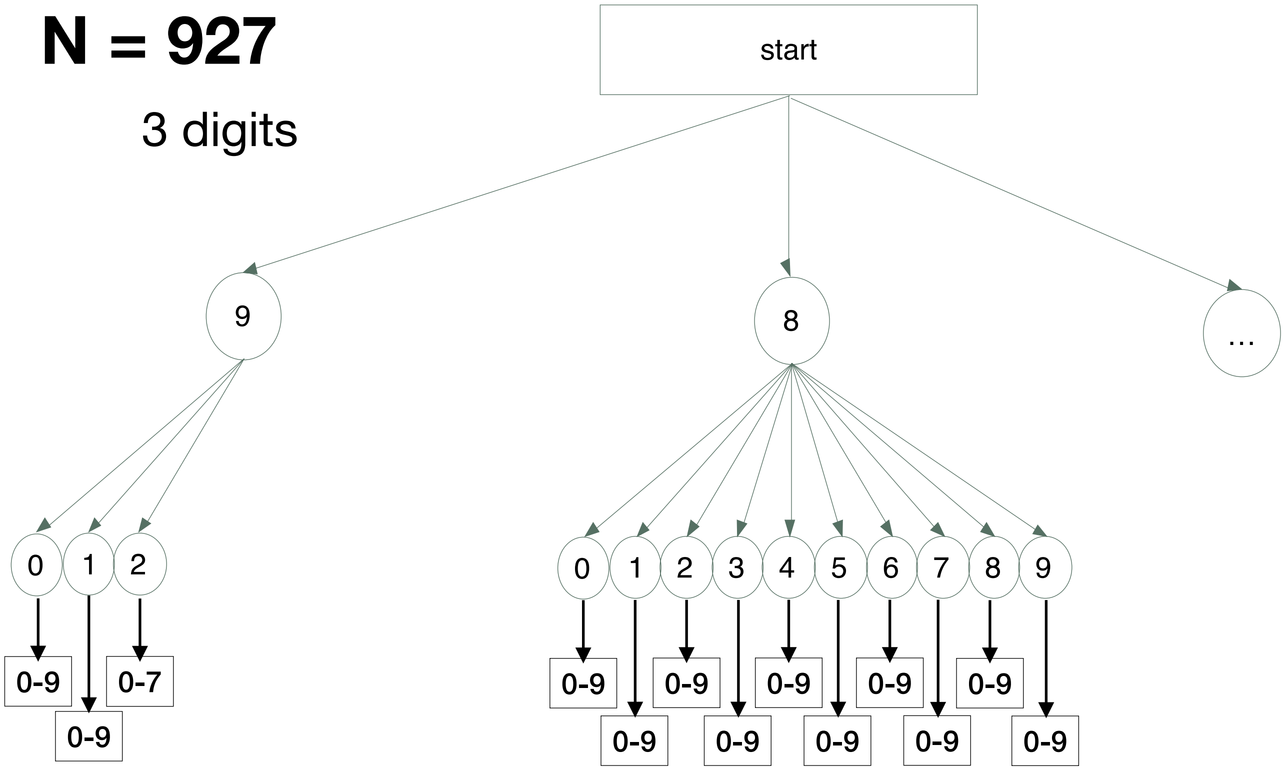
N = <u>927</u>

3 digits



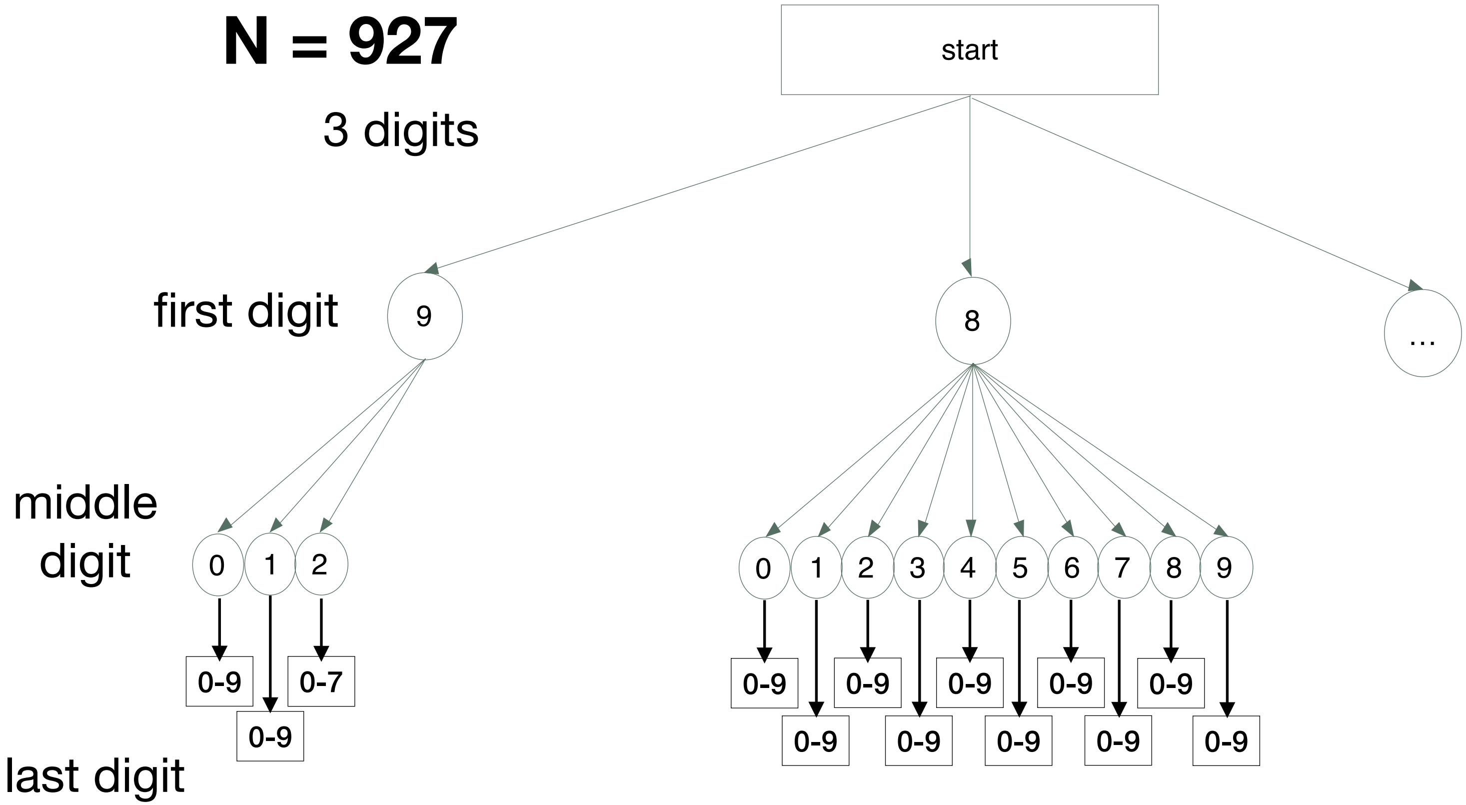
N = 927

3 digits



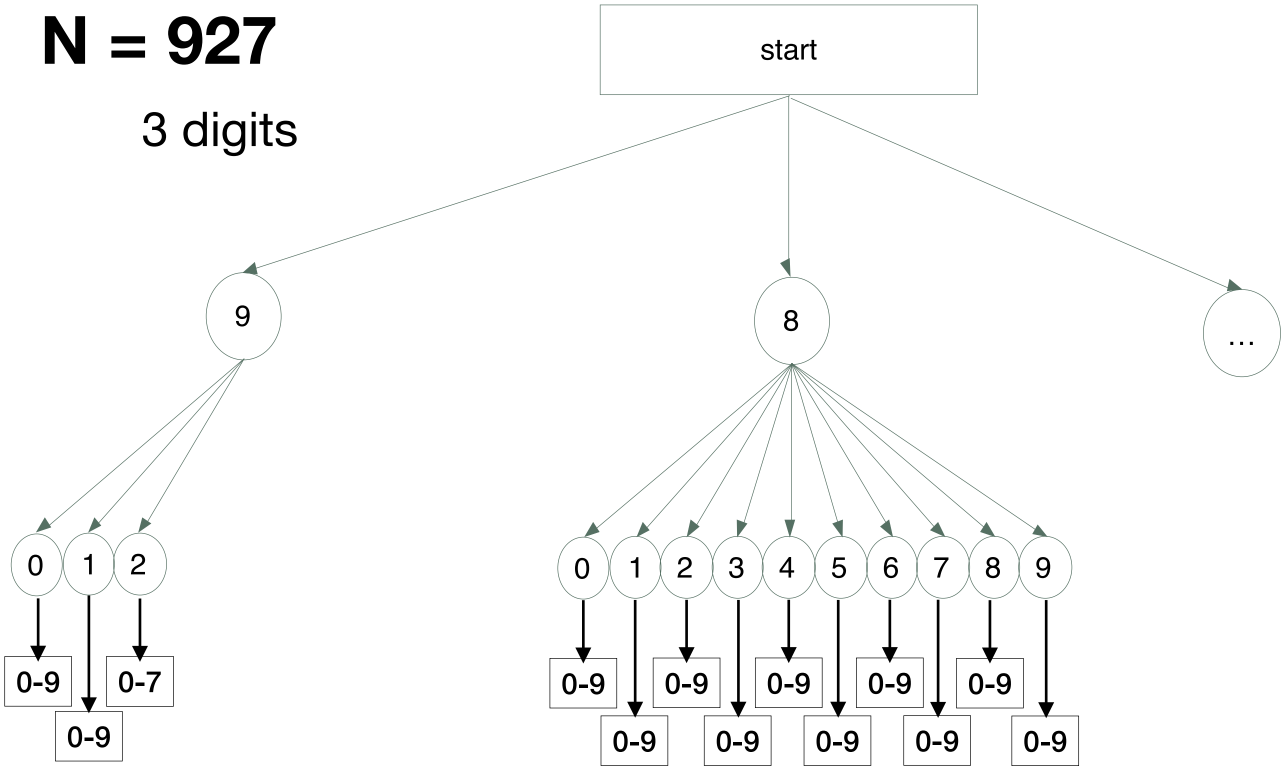
N = 927

3 digits



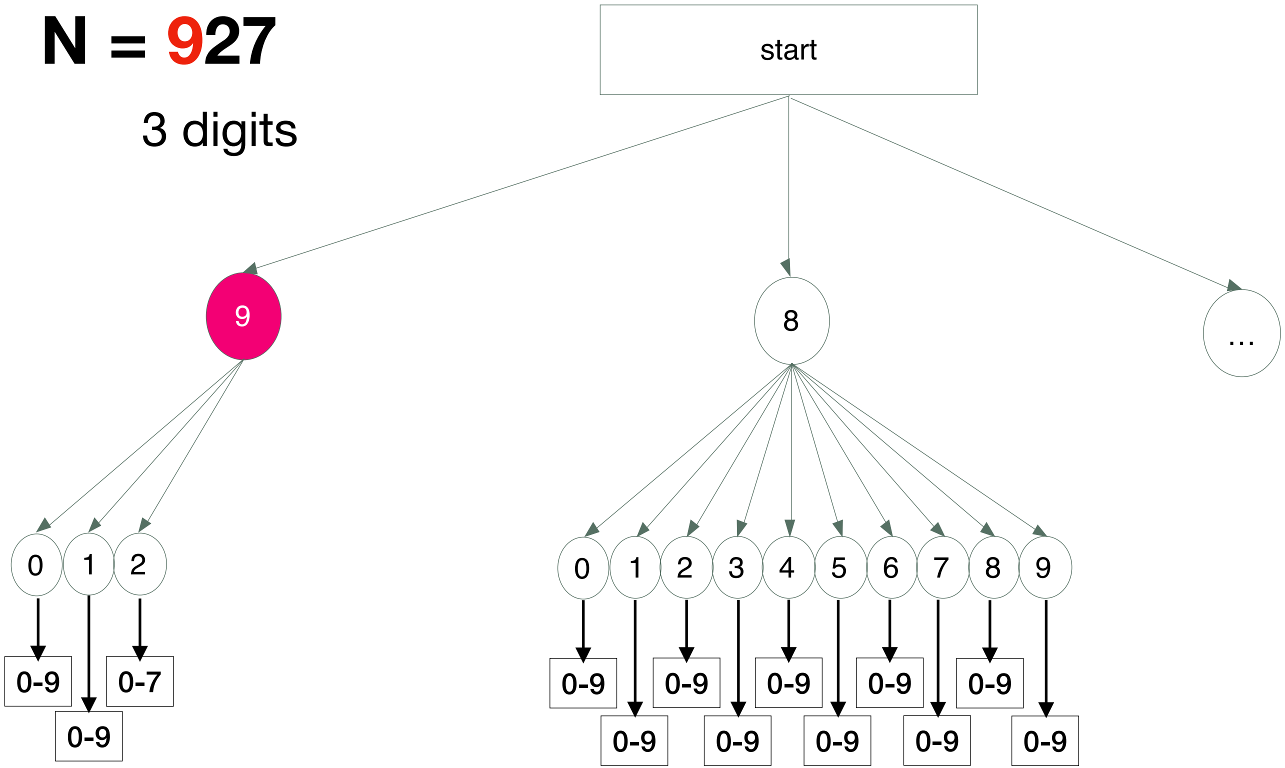
N = 927

3 digits



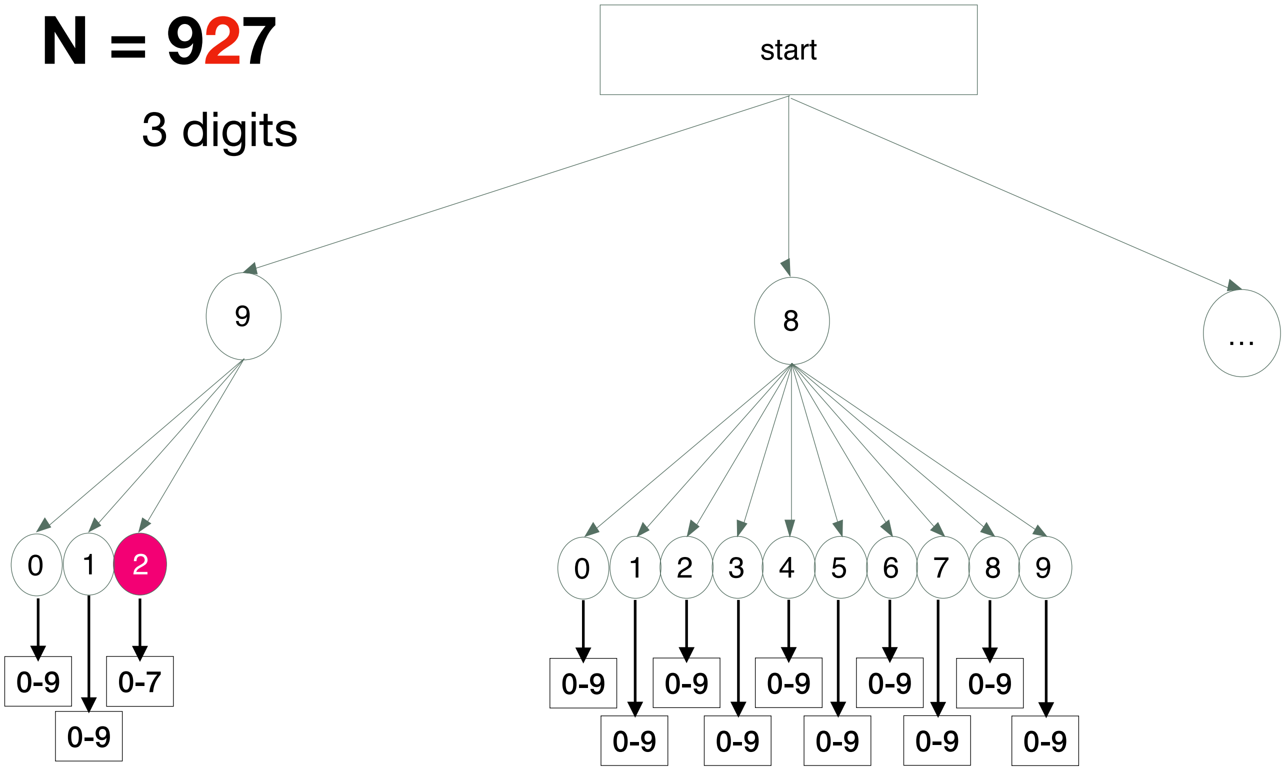
N = 927

3 digits



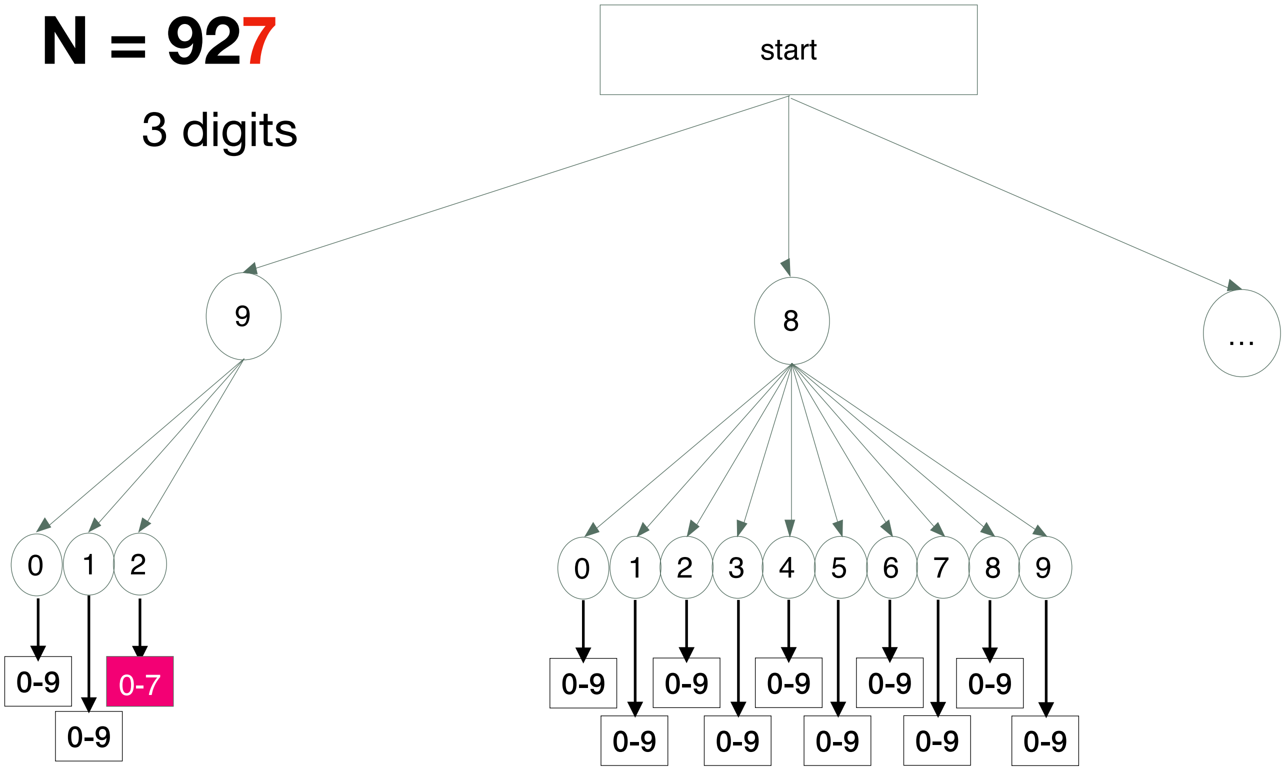
N = 927

3 digits



N = 927

3 digits



Solution

f(index, last_digit, same) = number of integers up to the i-th index which contain no two consecutive equal digits

Solution

f(index, last_digit, same)

Solution

f(index, last_digit, same)

Solution

f(index, last_digit, same)

same - whether the previous digits all match N

Solution

f(index, last_digit, same)

same - whether the previous digits all match N

f(index, last_digit, 1) = f(index - 1, N[index-1], 1) if (*N[index-1] != N[index]*)

f(index, last_digit, 0) = f(index - 1, N[index-1], 1) if (*N[index-1] != last_digit*
&& last_digit <= N[index])

+ f(index-1, j, 0) for all j != last_digit

Solution

f(index, last_digit, same)

same - whether the previous digits all match N

Case: same

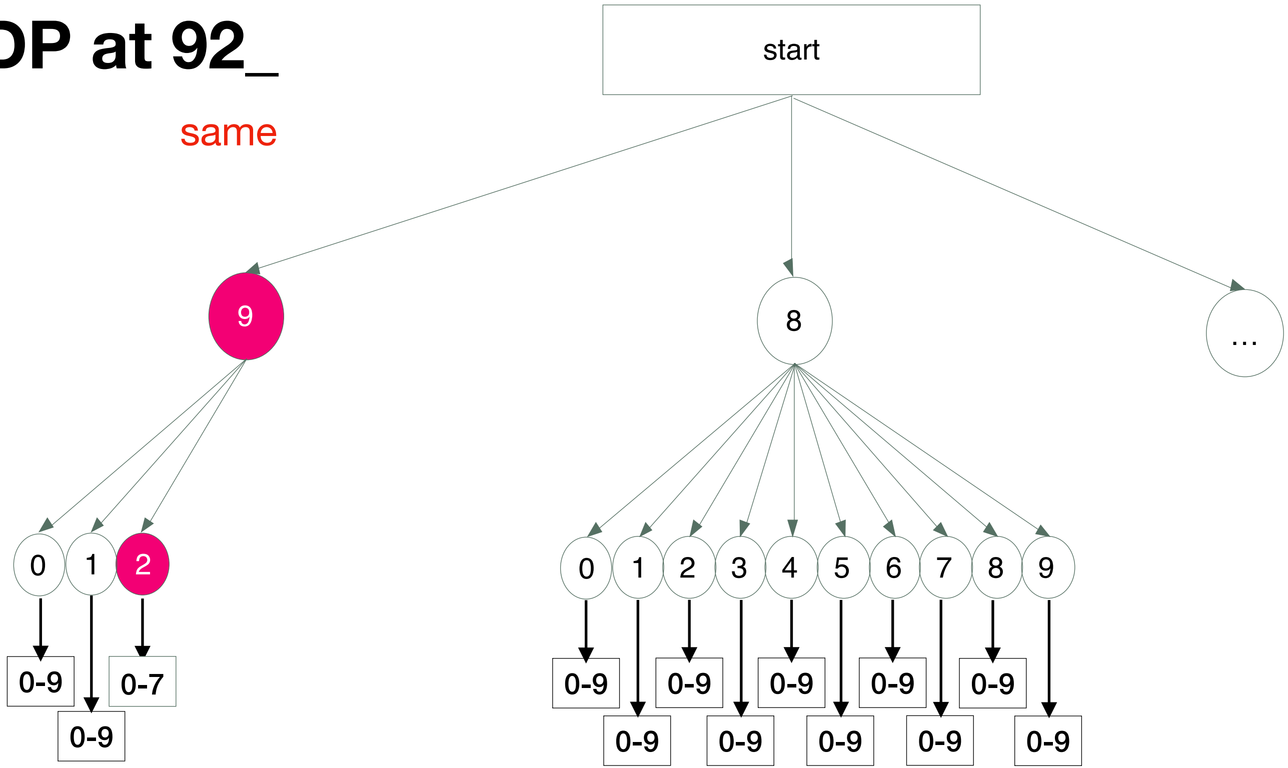
f(index, last_digit, 1) = f(index - 1, N[index-1], 1)   if (*N[index-1] != N[index]*)

f(index, last_digit, 0) = f(index - 1, N[index-1], 1)   if (*N[index-1] != last_digit*
&& last_digit <= N[index])

+ f(index-1, j, 0)   for all j != last_digit

N = 927
DP at 92_

same



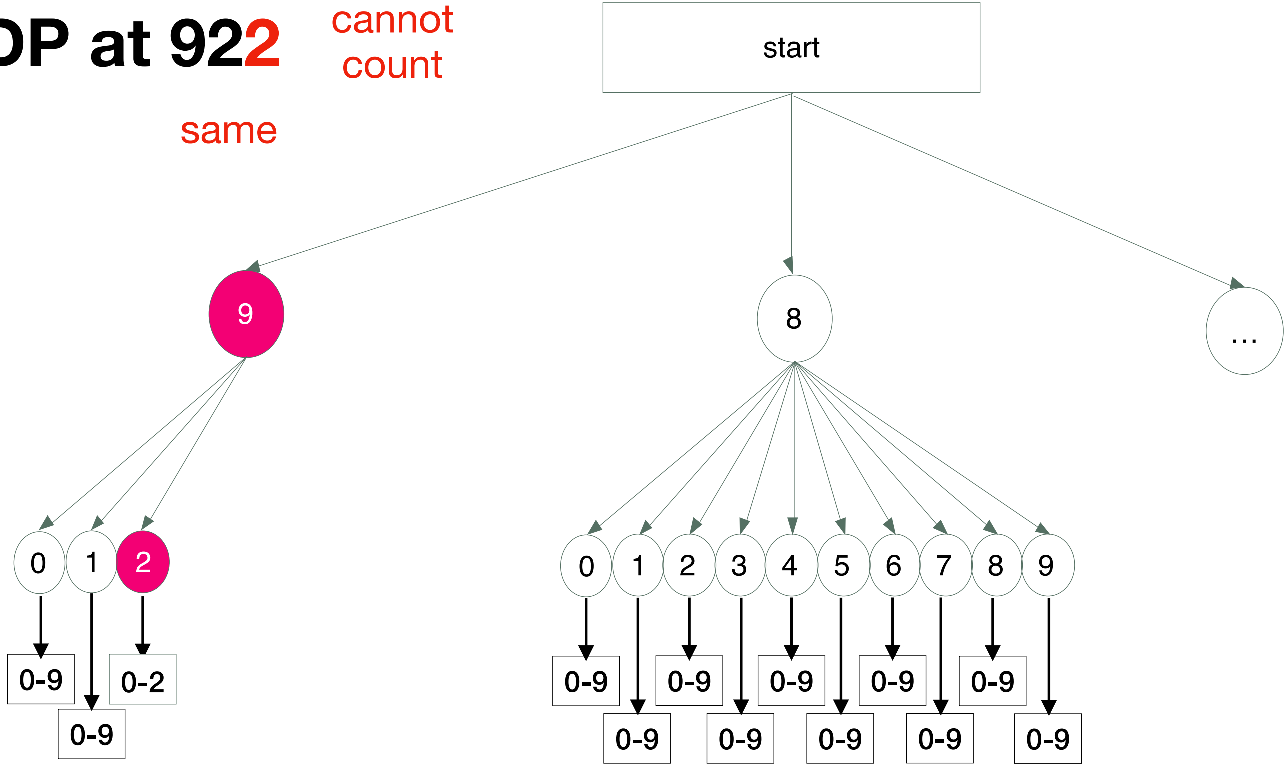
f(index, last_digit, 1) = f(index - 1, N[index-1], 1) if (*N[index-1] != N[index]*)

N = 922

DP at 922

cannot count

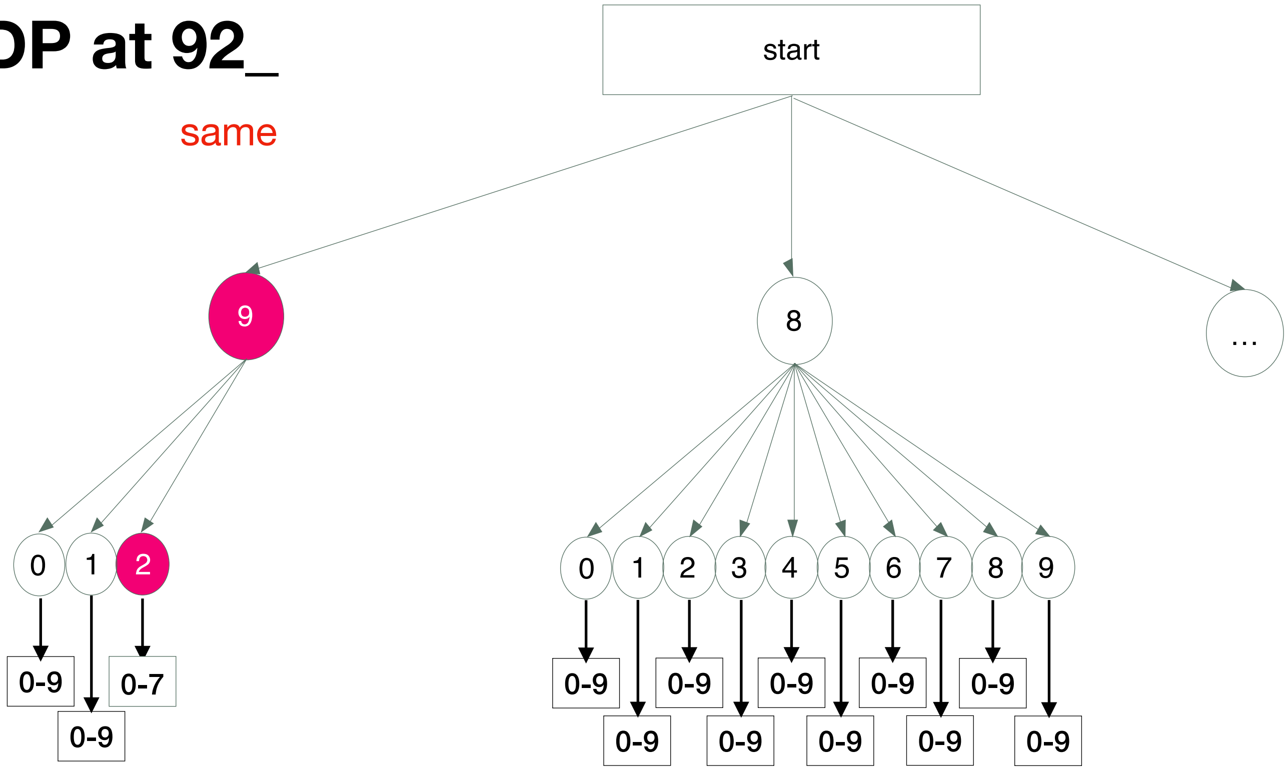
same



f(index, last_digit, 1) = f(index - 1, N[index-1], 1)   if (N[index-1] != N[index])

N = 927
DP at 92_

same



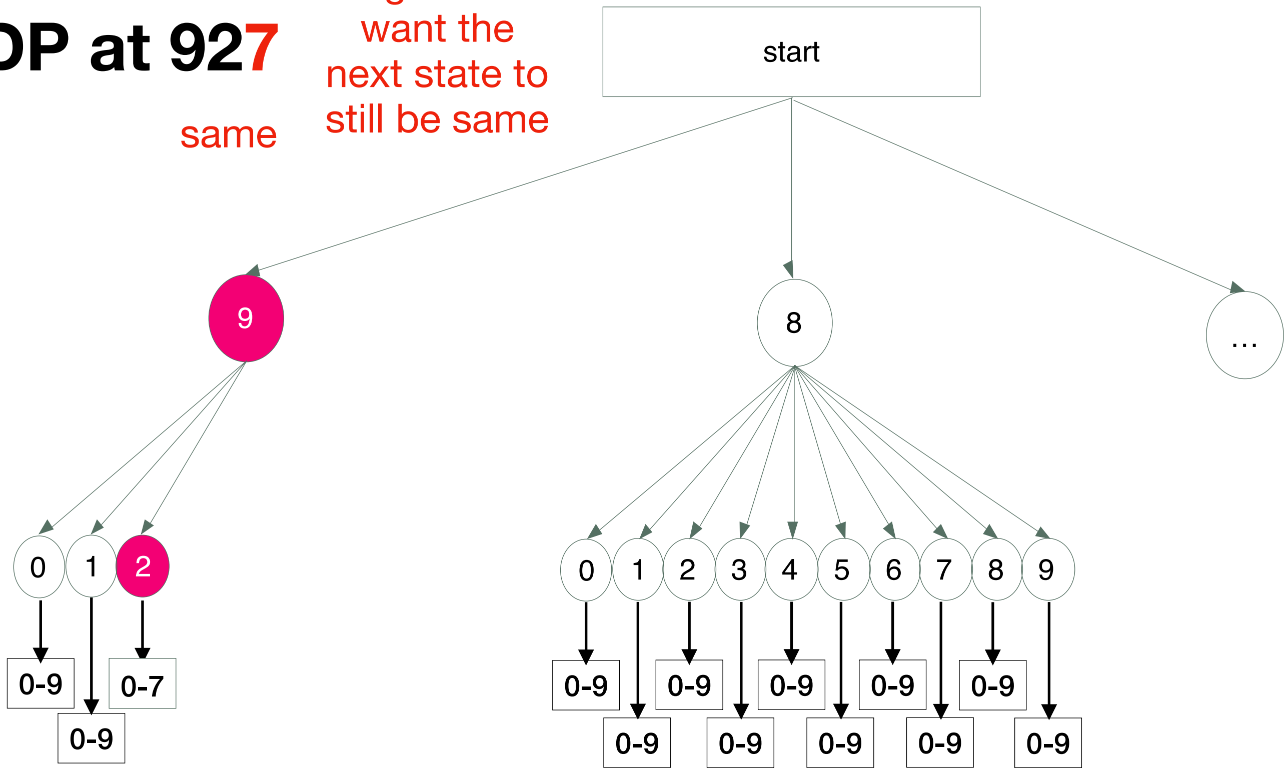
f(index, last_digit, 1) = f(index - 1, N[index-1], 1)   if (*N[index-1] != N[index]*)

N = 927

DP at 927

same

only possible digit if we want the next state to still be same



f(index, last_digit, 1) = f(index - 1, N[index-1], 1)   if (*N[index-1] != N[index]*)

Solution

f(index, last_digit, same)

same - whether the previous digits all match N

Case: same

f(index, last_digit, 1) = f(index - 1, N[index-1], 1) if (*N[index-1] != N[index]*)

f(index, last_digit, 0) = f(index - 1, N[index-1], 1) if (*N[index-1] != last_digit* && last_digit <= N[index])

+ f(index-1, j, 0) for all j != last_digit

Solution

f(index, last_digit, same)

same - whether the previous digits all match N

f(index, last_digit, 1) = f(index - 1, N[index-1], 1) if (*N[index-1] != N[index]*)

Case: not same

f(index, last_digit, 0) = f(index - 1, N[index-1], 1) if (*N[index-1] != last_digit* && last_digit <= N[index])

+ f(index-1, j, 0) for all j != last_digit

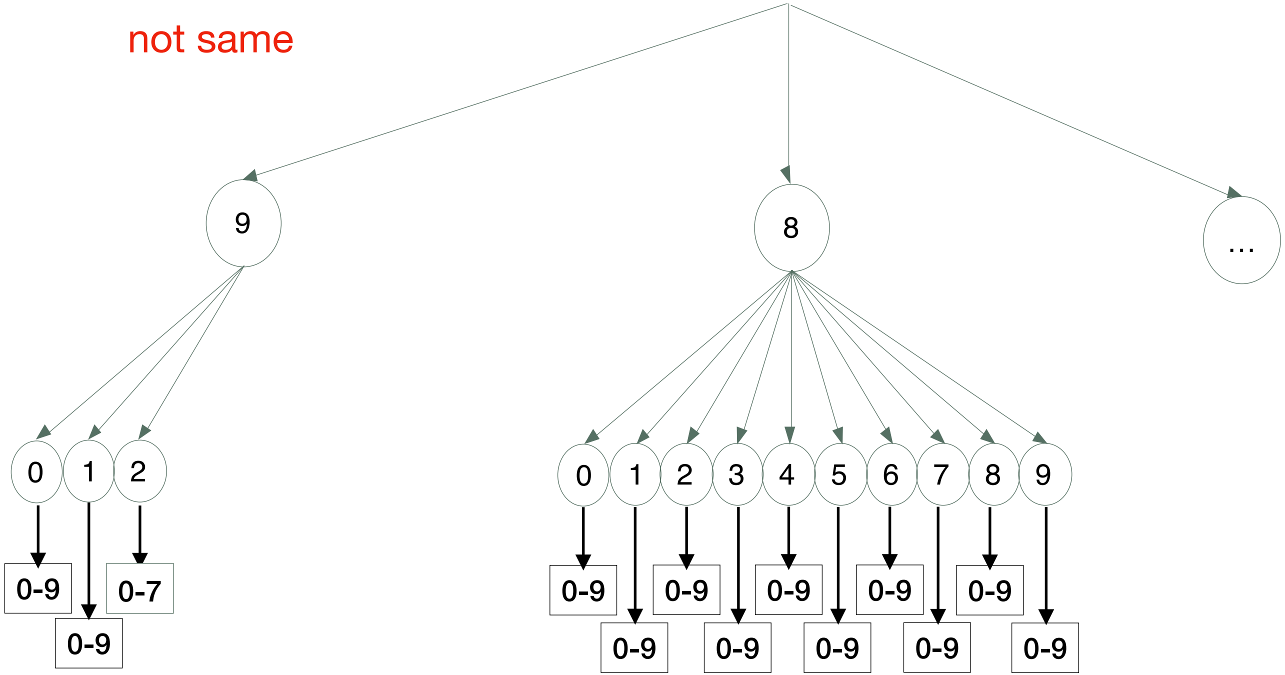
N = 927

DP at X__

not same

f(index, last_digit, 0) = f(index - 1, N[index-1], 1) if (*N[index-1] != last_digit* && last_digit <= N[index])

\+ f(index-1, j, 0) for all j != last_digit



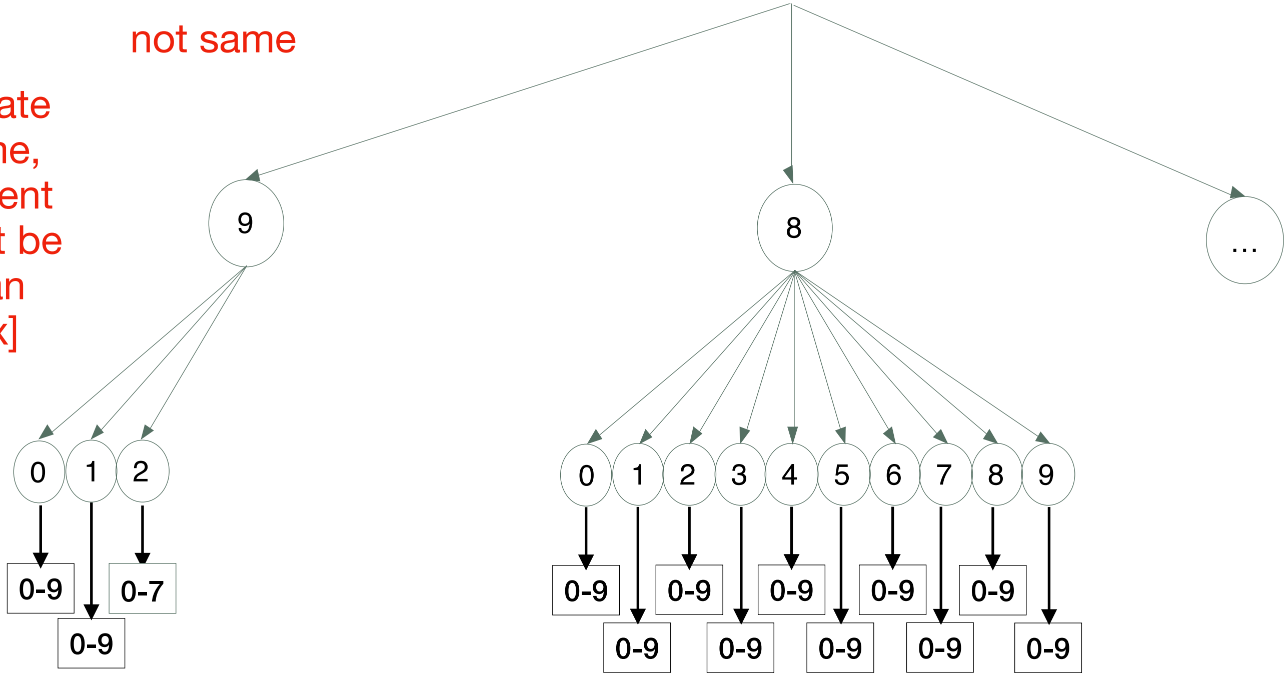
N = 927
DP at X__

not same

If prev state was same, then current digit must be less than N[index]

f(index, last_digit, 0) = f(index - 1, N[index-1], 1)   if (*N[index-1] != last_digit* && last_digit <= N[index])

+ f(index-1, j, 0)   for all j != last_digit



N = 927

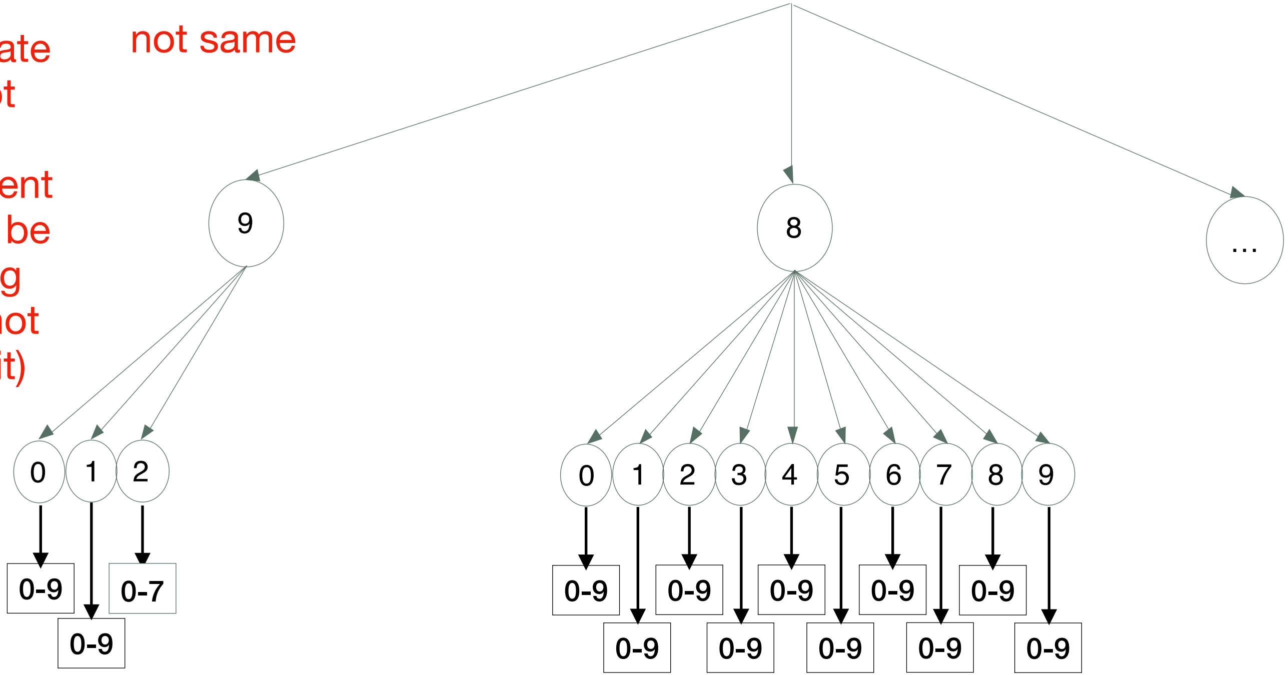
DP at X__

f(index, last_digit, 0) = f(index - 1, N[index-1], 1) if (*N[index-1] != last_digit* && last_digit <= N[index])

+ f(index-1, j, 0) for all j != last_digit

not same

If prev state was not same, then current digit can be anything (that is not last digit)



Solution

f(index, last_digit, same)

same - whether the previous digits all match N

f(index, last_digit, 1) = f(index - 1, N[index-1], 1)   if (*N[index-1] != N[index]*)

f(index, last_digit, 0) = f(index - 1, N[index-1], 1)   if (*N[index-1] != last_digit*
&& last_digit <= N[index])

+ f(index-1, j, 0)   for all j != last_digit

Solution

f(index, last_digit, same)

same - whether the previous digits all match N

f(index, last_digit, 1) = f(index - 1, N[index-1], 1)   if (N[index-1] != N[index])

f(index, last_digit, 0) = f(index - 1, N[index-1], 1)   if (N[index-1] != last_digit && last_digit <= N[index])

+ f(index-1, j, 0)   for all j != last_digit

no consecutive equal digits

Solution

f(index, last_digit, same)

same - whether the previous digits all match N

f(index, last_digit, 1) = f(index - 1, N[index-1], 1) if (<u>*N[index-1] != N[index]*</u>)

f(index, last_digit, 0) = f(index - 1, N[index-1], 1) if (<u>*N[index-1] != last_digit*</u> && last_digit <= N[index])

+ f(index-1, j, 0) for all j != last_digit

State: O(20 * lgN) = O(lg N)

Transition: O(10) = O(1)

Digit DP

Example: Numbers (BOI 2013)

https://oj.uz/problem/view/BOI13_numbers

Numbers (BOI 2013)

Problem Statement

- A string is a palindrome if it remains the same when it is read backwards.
- A number is palindrome-free if it does not contain a palindrome with a length greater than 1 as a substring.
- Your task is to calculate the total number of palindrome-free numbers in a given range: [a, b], $0 < a, b \leq 10^{18}$

Digit DP

Discuss!

Numbers (BOI 2013)

Solution

- Let the number of palindrome-free numbers from 1 to x be $f(x)$

Numbers (BOI 2013)

Solution

- Let the number of palindrome-free numbers from 1 to x be $f(x)$
- Number of palindrome-free numbers from a to b (inclusive): ***f(b) - f(a-1)***

Numbers (BOI 2013)

Solution

- Let the number of palindrome-free numbers from 1 to x be $f(x)$
- Number of palindrome-free numbers from a to b (inclusive): ***f(b) - f(a-1)***
- We just need to find ***f(b)*** and ***f(a-1)***

Numbers (BOI 2013)

Solution

- If a number consists a palindrome of length ***k*** and $k > 2$, then it consists a palindrome of length ***k* - 2** .

Numbers (BOI 2013)

Solution

- If a number consists a palindrome of length ***k*** and $k > 2$, then it consists a palindrome of length ***k* - 2** .
- Eg: A palindrome of length 5 consists a palindrome of length 3.
- Note: A single digit isn't considered a palindrome (stated in the question)

Numbers (BOI 2013)

Solution

- If a number consists a palindrome of length ***k*** and $k > 2$, then it consists a palindrome of length ***k - 2*** .
- Eg: A palindrome of length 4 consists a palindrome of length 2.
- Note: A single digit isn't considered a palindrome (stated in the question)

Numbers (BOI 2013)

Solution

- To check for the existence of palindromes of *any* length,
- *We just need to check for palindromes of length **2** and **3**.*
- We only need to care about any ***3 consecutive digits***!

Numbers (BOI 2013)

Solution

- To check for the existence of palindromes of *any* length,
- *We just need to check for palindromes of length **2** and **3**.*
- We only need to care about any ***3 consecutive digits***!

```
dp (prev1, prev2, index, same) {
    //Complete this code yourself
}
```

Lexicographical DP

Lexicographical DP

Examples:

- A string can contain [set of letters]
- A string is good if [some condition]
    - Find k-th lexicographically smallest string
    - Find lexicographical index of a string

Kth lexicographically smallest string

DP - number of strings satisfying **suffix**

```
f(index, last_letter){
    if(index==intended length) return 1;
    int result=0;
    for(newletter in set of accepted letters)
        result+=f(index+1, newletter);
    return result;
}
```

Kth lexicographically smallest string

Build up string letter by letter

Keep count of number of strings already smaller than current string

Update count using precomputed DP

Pick the largest letter such that count ≤ K

Kth lexicographically smallest string

```
int count = 0;
char ans[N];

for (int i=0; i<N; i++) {
    find largest letter where count + dp(i, letter-1) < K
    count += dp(i, letter-1);
    ans[i] = letter;
}
```

Lexicographical index of string

Go letter by letter

Count number of strings strictly smaller than current suffix

```
int rank = 0;
for (int i=0; i<N; i++) {
    rank += dp(i, str[i]);
}
```

Let’s try some problems!

Raymondland

Raymondland

Problem Statement

Raymondland is very peculiar. Everyone is very superstitious. When Marianna visited Raymondland, she realized that certain floors are ‘missing’ from the hotel building—numbers containing 4 and 13 as substrings are omitted from the floor numberings. This is because 4 and 13 are considered unlucky numbers.

Raymondland

Problem Statement

For simplicity, we will refer to this numbering scheme as the lucky numbering scheme, as it omits the unlucky numbers.

Raymondland

Problem Statement

The table shows the first 20 floors in a lucky numbering scheme as well as the conventional numbering scheme.

Conventional	Lucky
1	1
2	2
3	3
4	5
5	6
6	7
7	8
8	9
9	10
10	11

Conventional	Lucky
11	12
12	15
13	16
14	17
15	18
16	19
17	20
18	21
19	22
20	23

Raymondland

Problem Statement

Marianna thinks that Raymondland is weird and wants to be able to convert floors between the lucky and conventional numbering scheme. Hence, given a floor number in the lucky numbering scheme, Marianna wants you to compute which floor it will be in the conventional numbering scheme and vice-versa.

Raymondland

Constraints

There will be **N ≤** 100000 floor numbers **X[i]** to be converted (from either lucky to conventional or conventional to lucky).

X[i] ≤ 10^16.

4 seconds, 256 MB.

Raymondland

Discuss!

10^16 if-else statements

Haha

10^16 if-else statements

Haha

No

String processing

To check if a number is unlucky:

1. Convert the number into individual digits (using sprintf/stringstream/base conversion)
2. Loop through each digit to check for presence of ‘4’
3. Loop through each consecutive pair of digits to check for presence of ’13’

Loop through from 1 onwards, count the number of unlucky numbers to calculate the answer

String processing

```
bool unlucky(long long x) {
    char str[20];
    sprintf(str, "%lld", x);
    for (int i = 0, l = strlen(str); i < l; i++) {
        if (str[i] == '4') return 1;
        if (i == 0) continue;
        if (str[i-1] == '1' && str[i] == '3') return 1;
    }
    return 0;
}
```

String processing

```
bool unlucky(long long x) {
    char str[20];
    sprintf(str, "%lld", x);
    for (int i = 0, l = strlen(str); i < l; i++) {
        if (str[i] == '4') return 1;
        if (i == 0) continue;
        if (str[i-1] == '1' && str[i] == '3') return 1;
    }
    return 0;
}
```

Works till maybe X[i] ≤ 1,000,000

String processing + Counter

Maintain 2 counters:

1. Counter for conventional
2. Counter for lucky

String processing + Counter

Maintain 2 counters:

1. Counter for conventional
2. Counter for lucky

```
for (int c = 1, l = 1; c <= 100000; c++, l++) {
    while (unlucky(l)) l++;
    lucky_to_conv[l] = c;
    conv_to_lucky[c] = l;
}
```

Find Formula

k	10^k-1	f(k)	
1	9	8	
2	99	79	
3	999	710	(79-8)*10
4	9999	6318	(710-79)*10+8
5	99999	56159	(6318-710)*10+79
6	999999	499120	(56159-6318) *10+710
k	10^k-1	f(k) =	(f(k-1) – f(k-2))*10 + f(k-3)

Find Formula

Haha

Find Formula

Haha

No

DP

DP

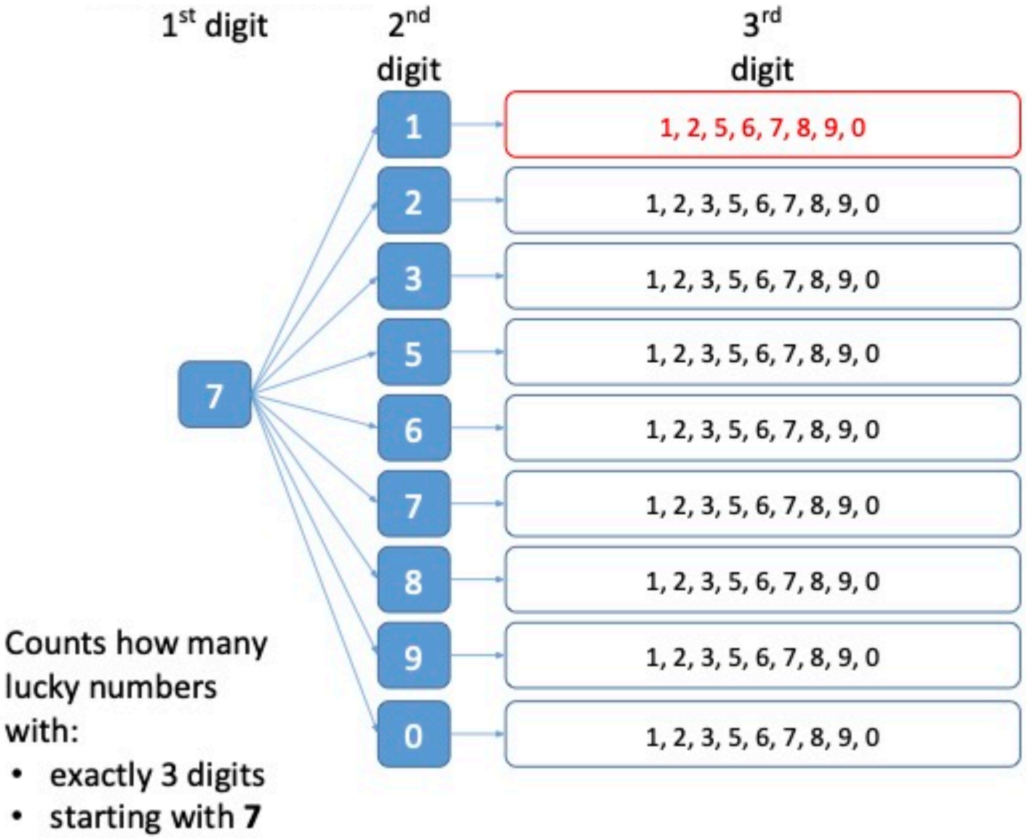
Surprise!

DP

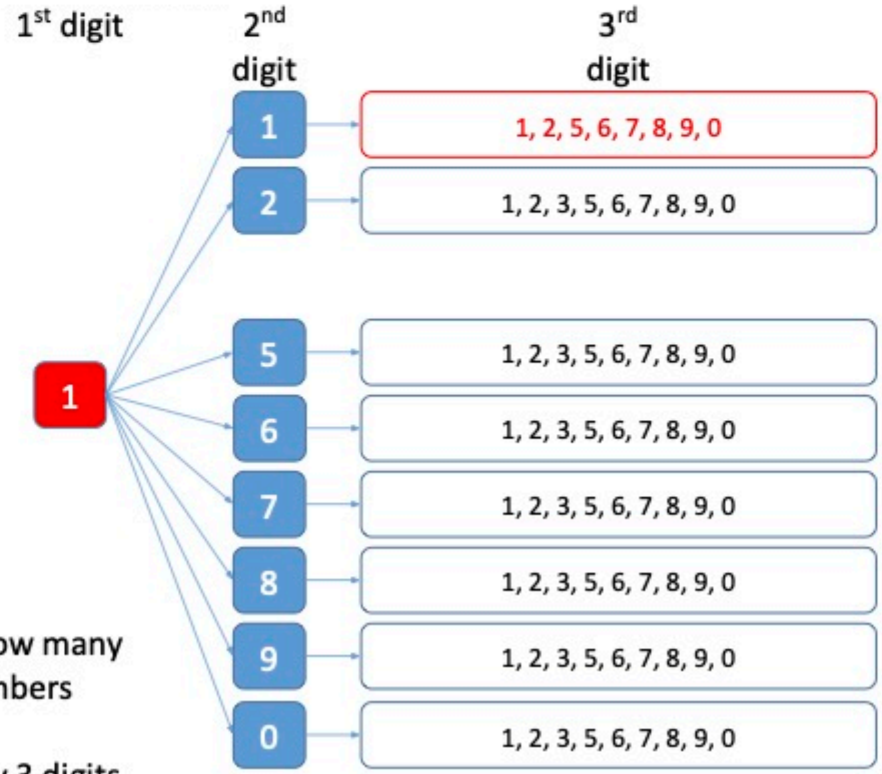
Surprise!

Case 1: Lucky to Conventional

Digit DP

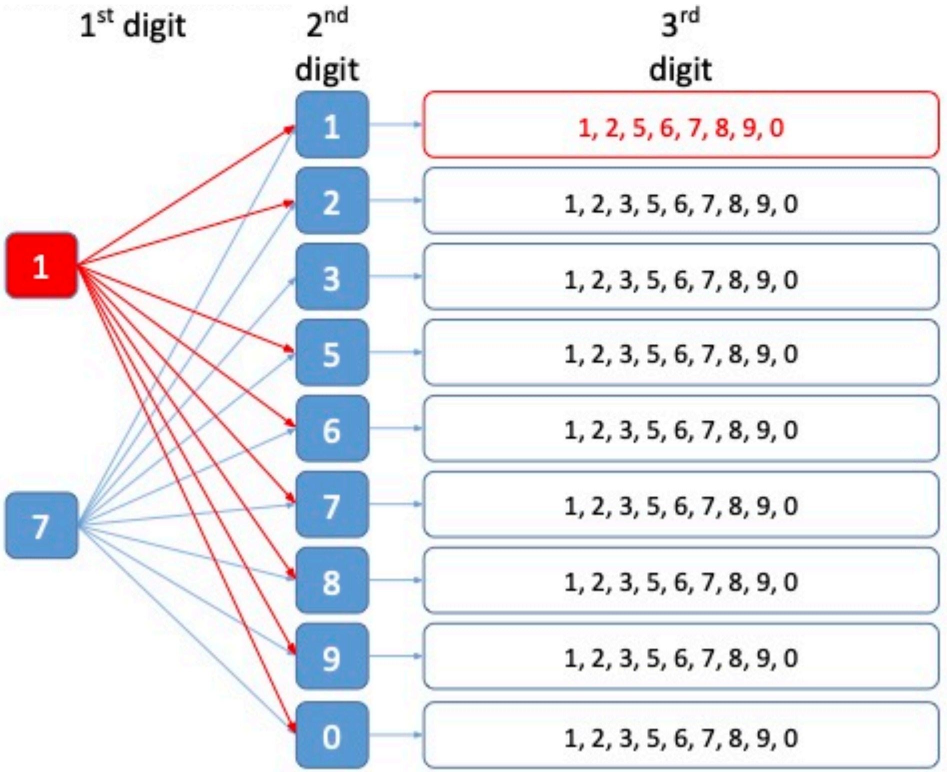


1st digit
2nd digit
3rd digit
7
1
1, 2, 5, 6, 7, 8, 9, 0
2
1, 2, 3, 5, 6, 7, 8, 9, 0
3
1, 2, 3, 5, 6, 7, 8, 9, 0
5
1, 2, 3, 5, 6, 7, 8, 9, 0
6
1, 2, 3, 5, 6, 7, 8, 9, 0
7
1, 2, 3, 5, 6, 7, 8, 9, 0
8
1, 2, 3, 5, 6, 7, 8, 9, 0
9
1, 2, 3, 5, 6, 7, 8, 9, 0
0
1, 2, 3, 5, 6, 7, 8, 9, 0
Counts how many lucky numbers with:
• exactly 3 digits
• starting with 7

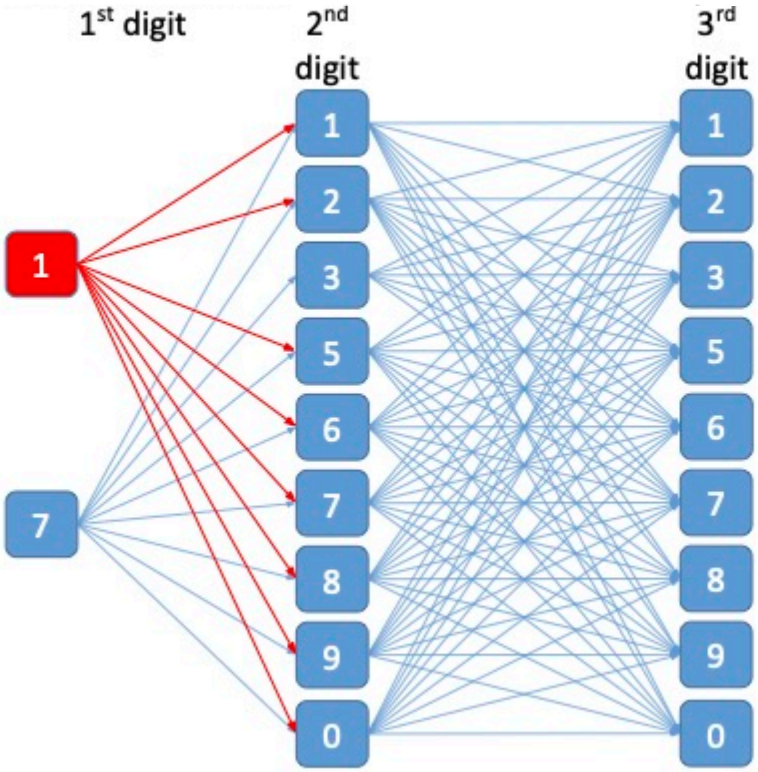


Counts how many lucky numbers with:

- exactly 3 digits
- starting with **1**



Overlapping subproblems!



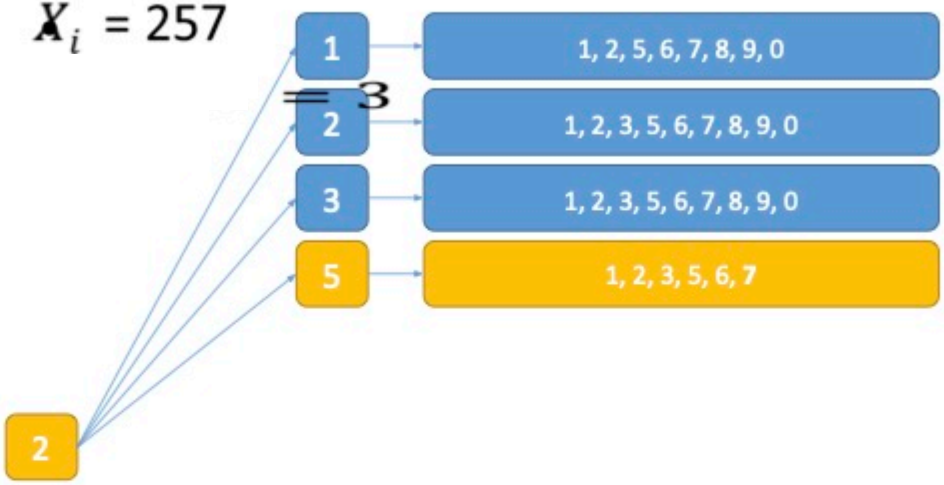
Overlapping subproblems!

$X_i = 257$

Counts how many lucky numbers with:

- exactly 3 digits
- starting with 1

$X_i = 257$

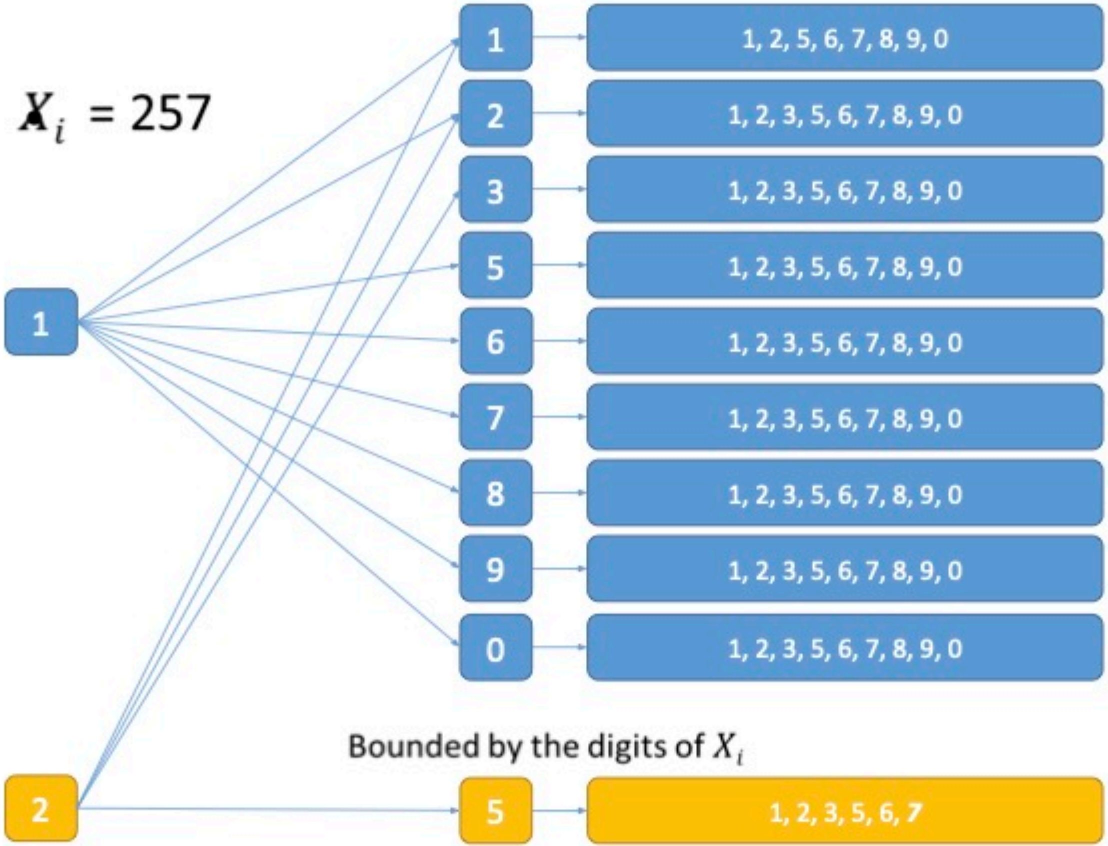


Counts how many lucky numbers with:

- exactly 3 digits
- starting with **2**
- Not more than X_i

Counts how many lucky numbers with:

- exactly 3 digits
- starting with **25**
- Not more than X_i



X_i = 257
1
2
1
2
3
5
6
7
8
9
0
1, 2, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
1, 2, 3, 5, 6, 7, 8, 9, 0
Bounded by the digits of X_i
5
1, 2, 3, 5, 6, 7

DP

Case 1: Lucky to Conventional

Digit DP

DP

Case 1: Lucky to Conventional

Digit DP

Case 2: Conventional to Lucky

DP

Case 1: Lucky to Conventional

Digit DP

Case 2: Conventional to Lucky

???

DP

Case 1: Lucky to Conventional

Digit DP

Case 2: Conventional to Lucky

BSTA + Case 1

BSTA

Binary Search the Answer

BSTA

Binary Search the Answer

- Guess a certain floor in the lucky numbering scheme

BSTA

Binary Search the Answer

- Guess a certain floor in the lucky numbering scheme
    - Convert it to conventional to check

BSTA

Binary Search the Answer

- Guess a certain floor in the lucky numbering scheme
  - Convert it to conventional to check
  - How? Verify using Case 1's solution

  - (implementation details: inputs up to 7.7*1017 = 18 digits total)

BSTA

Binary Search the Answer

- Guess a certain floor in the lucky numbering scheme
    - Convert it to conventional to check
    - If guess is too high, the answer must be lower

BSTA

Binary Search the Answer

- Guess a certain floor in the lucky numbering scheme
    - Convert it to conventional to check
    - If guess is too high, the answer must be lower
    - If guess is too low, the answer must be higher

More speeding up

If you look cloooosely, more state manipulation is possible but we won't talk about it.

More speeding up

If you look clooooosely, more state manipulation is possible but we won't talk about it.

Current Runtime:
O(N * states * avg. transition * log10^18)

Estimated iterations:
100000 * (18 * 10 * 2) * 10 * 60 = 2.16 x 10^10 iterations

DP II Mashup