Dynamic Programming 2

Problems we will look at

1. Change making problem

2. Edit distance problem

Change Making problem

In this problem, our goal is to find the minimum number of coins that add up to the given amount of money.

- We have to pick coins from a given denomination (e.g.: { 1, 5, 10, 25 })
- The coins can be reused.
- Sum should be exactly equal to a given amount N.

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Target sum: 12

Possible options below.

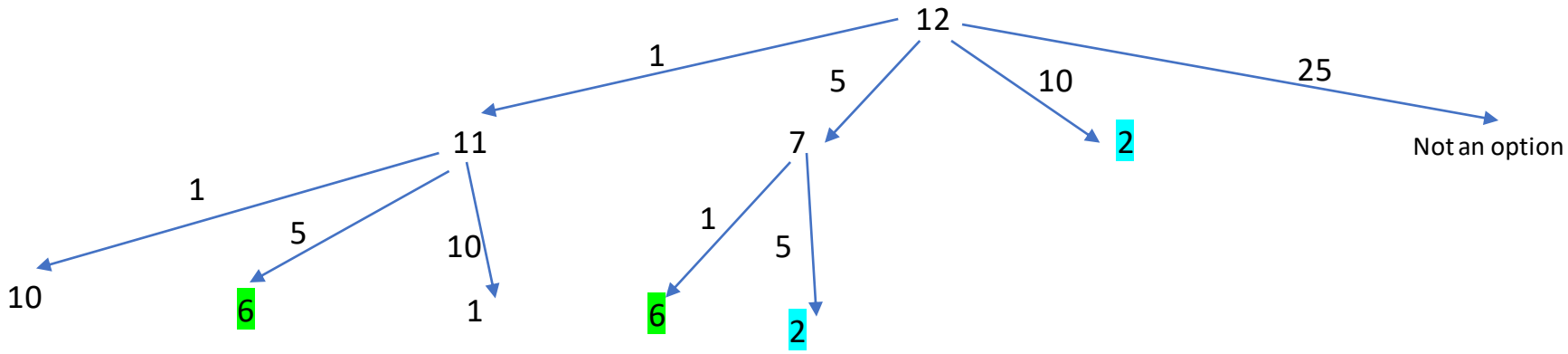
This is the one using minimum number of coins.

- 1 + 1 + 1 + … 12 times
- 5 + 1 + 1 + … 7 times
- 5 + 5 + 1 + 1
- 10 + 1 + 1

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Target sum: 12



- We see the overlapping subproblems here and here
- If we draw more levels of the tree, we will see more overlapping subproblems.
- You can imagine that the optimal solution will be the subtree with the least height.
- Final optimal solution (min # of coins): Combination of optimal solutions of sub problems (More on next slide).
- Hence, we should use DP.

- Recurrence relation and time/space complexity and code (next few slides) …

Change Making problem

From previous slide:

Final optimal solution (min # of coins): Combination of optimal solutions of sub problems

Let's say we have the optimal solution to sum up to N using the given coins { c1, c2, c3, c4 }
Now, lets break that solution sequence into two parts, the *left part* and *right part*.

c1	c4	c3	c1	c2	c4	c4	c3

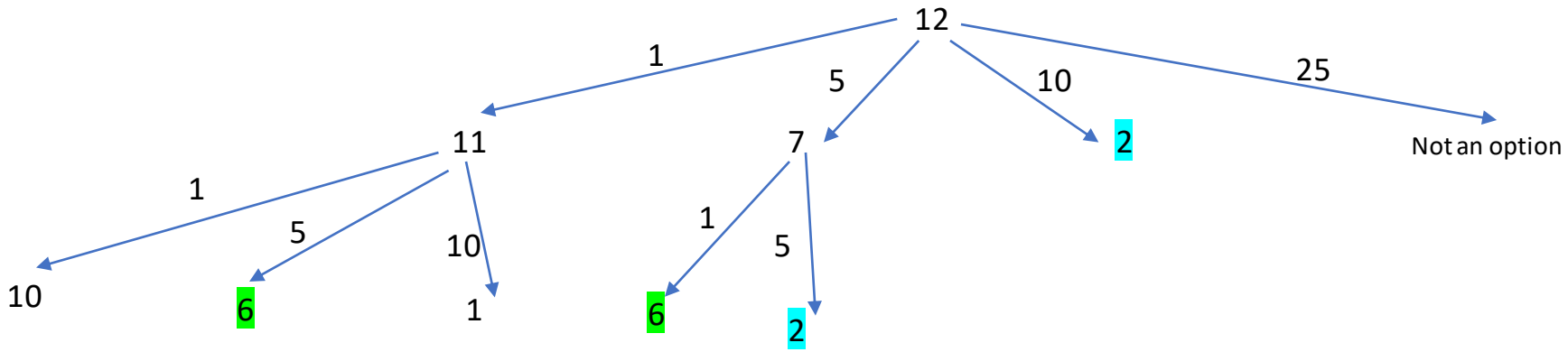
- left part sums up to N-K
- right part sum up to K

- c1 + c4 + c3 + c1 + c2 + c4 = N - K
- c4 + c3 = K

The solutions for left part and right part must be the optimal solutions for N-K and K respectively, for the whole solution to be optimal for a sum of N.

Change Making problem

Recurrence relation:



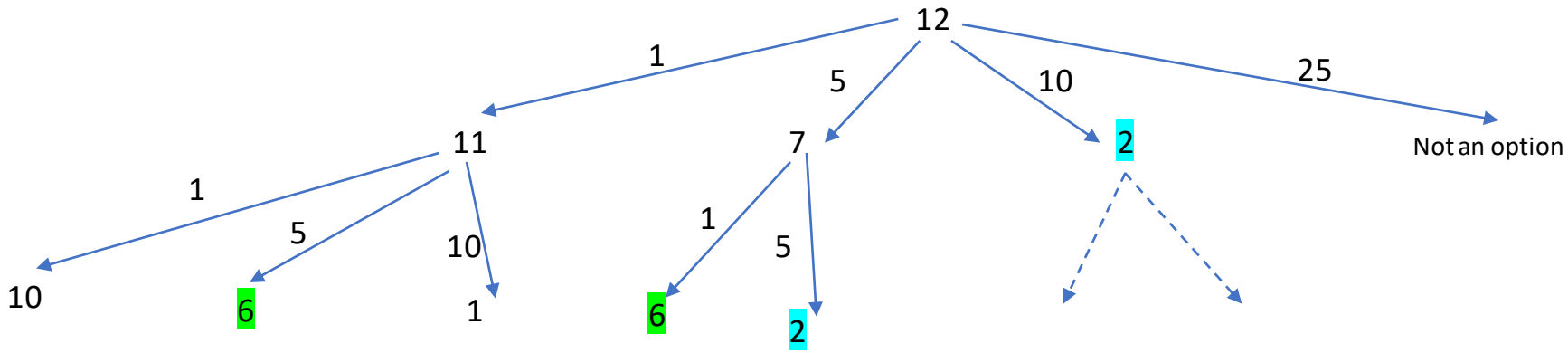
NumCoins (N) = ∀ given coin denominations i (1 + Min (NumCoins (N – Ci))) ← See explanation next

This is the coin Ci

Residual value after using coin Ci

Change Making problem

Recurrence relation:



NumCoins (N) = ∀ given coin denominations i (1 + Min (NumCoins (N – Ci)))

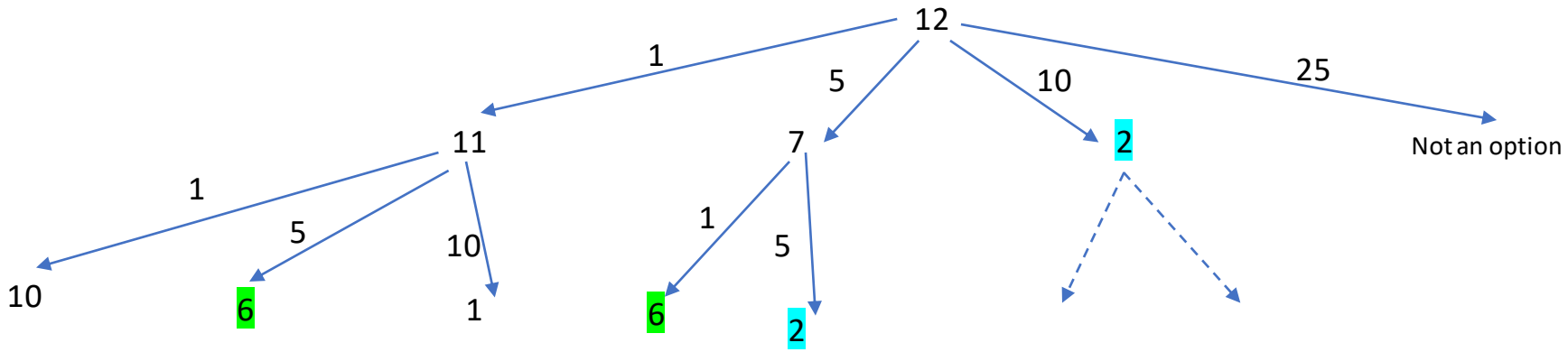
Say, N = 10.

Now, if I pick:

- 1 cent coin: NumCoins(10) = 1 + Min (NumCoins(9))

Change Making problem

Recurrence relation:



NumCoins (N) = ∀ given coin denominations i (1 + Min (NumCoins (N – Ci)))

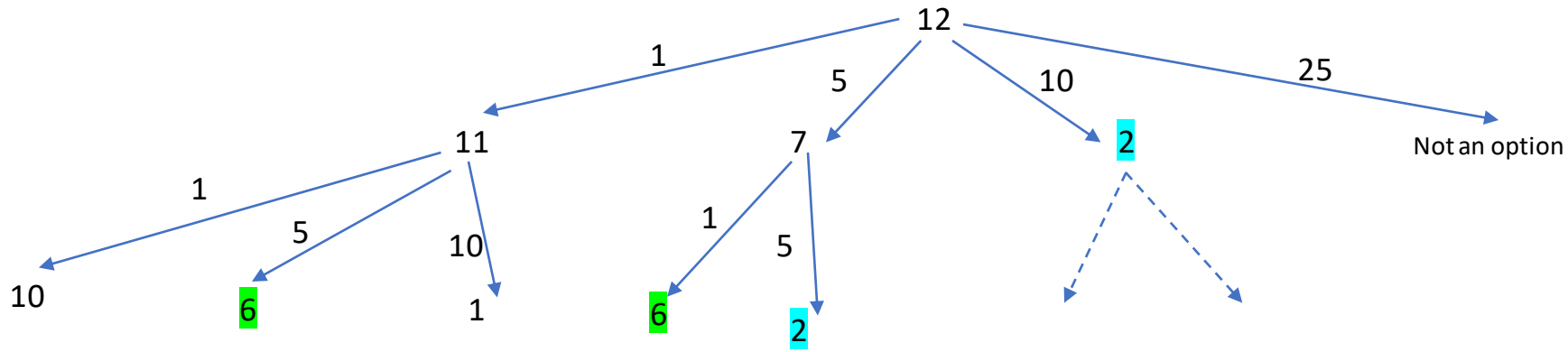
Say, N = 10.

Now, if I pick:

- 1 cent coin: NumCoins(10) = 1 + Min (NumCoins(9))
- 5 cent coin: NumCoins(10) = 1 + Min (NumCoins(5))

Change Making problem

Recurrence relation:



NumCoins (N) = ∀ given coin denominations i (1 + Min (NumCoins (N – Ci)))

Say, N = 10.

Now, if I pick:

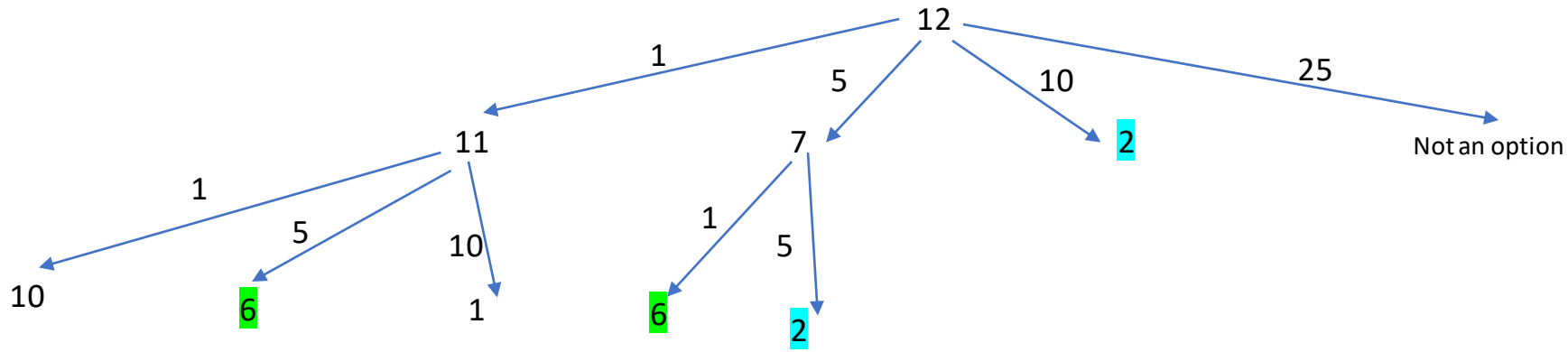
Question: What r time and space complexity?
info on next slide

- 1 cent coin: NumCoins(10) = 1 + Min (NumCoins(9))
- 5 cent coin: NumCoins(10) = 1 + Min (NumCoins(5))
- 10 cent coin: NumCoins(10) = 1 + Min (NumCoins(0))

I want the min of these 3 options.

Change Making problem

Recurrence relation:



NumCoins (N) = ∀ given coin denominations i (1 + Min (NumCoins (N – Ci)))

Time: Exponential $O(|C|^N)$ ← number of branches at each level: 1st level: 1, 2nd level: C, 3rd level: C^2 Nth level: C^N

Note: We will have N levels if we have a 1 cent coin. Or, in general, N/C_{min} levels

Space: O (N) due to recursion

Change Making problem

Recursive solution without DP

```
int GetMinCoins ( int N )

        if N <= 0
                return 0

        minCoins = MAX_INT


        foreach   coin                    ← loop runs C times
                if ( coin > N )                              // coin is too big, cant use
                        continue

                residualValue  =   N – coin
                minCoins  =  MIN (   1  +  GetMinCoins ( residualValue ),   minCoins  )


        return   minCoins
```

Time: Exponential $O(C^N)$

Space: O (N) due to recursion

Change Making problem

Adding memoization:

```
int GetMinCoins ( int N )
        if N <= 0
                return 0

        if (  cache[ N ]  >= 0 )
                return  cache [ N ]

        minCoins = MAX_INT

        foreach   coin                          ← loop runs C times
                if  ( coin > N )                                    // coin is too big, cant use
                        continue

                residualValue =   N – coin
                minCoins  =  MIN (  1  +  GetMinCoins ( residualValue ),   minCoins  )

        cache [ N ] = minCoins

        return minCoins
```

Time complexity: [redacted]

Space complexity: [redacted]

Change Making problem

Adding memoization:

```
int GetMinCoins ( int N )
        if N <= 0
                return 0

        if (  cache[ N ]  >= 0 )
                return  cache [ N ]

        minCoins = MAX_INT

        foreach   coin                          ← loop runs C times
                if  ( coin > N )                                    // coin is too big, cant use
                        continue

                residualValue =   N – coin
                minCoins  =  MIN (  1  +  GetMinCoins ( residualValue ),   minCoins  )

        cache [ N ] = minCoins

        return minCoins
```

Time complexity: O(N * C)

Space complexity: O (N) due to cache

Change Making problem

Let’s look at the bottom-up approach.

Here’s the table in our tabulation approach.

Sum	0	1	2	3	4	5	6	7
Num coins								

Coin denomination given: { 1, 5, 10, 25 }

What is our initial condition ?

- What cell should we fill up?
- And with what value?

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0							

What is our initial condition ?

- What cell should we fill up? First cell under value 0
- And with what value? We put 0. No coins needed for a sum of 0

What next?

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1						

Next:

Fill up num coins needed for a sum of 1 … the only denomination that works is 1… and we need one of that coin.

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	?					

Next:

Fill up num coins needed for a sum of 2 … the only denomination that works is 1.

So, we will say : " I need one of 1cent coin", and the residual value will be 2 – 1 = 1 …. And for that residual value, I will simply look at my table (bottom up, so I must have already calculated for residual value) …

So, I will look at cache [residual value] ➔ cache [1]

In other words: 1 + cache[2 – 1]

= 1 + cache [1]

= 1 + 1 = 2

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	1+1 = 2					

We put 2 coins for a sum of 2.

Next:

Fill up num coins needed for a sum of 3 … the only denomination that works is 1.
So, we will say : "I need one of 1cent coin", and the residual value will be 3 – 1 = 2
And look up entry for residual value

In other words: 1 + cache[3 – 1] = 1 + cache [2] = 1 + 2 = 3

Q: What if we had a coin denomination of 2?

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	1+1 = 2					

We put 2 coins for a sum of 2.

Next:

Fill up num coins needed for a sum of 3 … the only denomination that works is 1.
So, we will say : "I need one of 1cent coin", and the residual value will be 3 – 1 = 2
And look up entry for residual value

In other words: 1 + cache[3 – 1] = 1 + cache [2] = 1 + 2 = 3

This would be the 2 cent coin

Q: What if we had a coin denomination of 2? ➔ 1 + cache[3 – 2] = 1 + cache [1] = 1 + 1 = 2

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	1+1 = 2	1+2 = 3				

Next:

Fill up num coins needed for a sum of 4 … the only denomination that works is 1.
So, we will say : "I need one of 1cent coin", and the residual value will be 4 – 1 = 3

1 + cache[4 – 1] = 1 + cache [3] = 1 + 3 = 4

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	1+1 = 2	1+2 = 3	1+3 = 4			

Fill up num coins needed for a sum of 5 … the denominations that work are 1 and 5

For 1 cent:

"I need one of 1cent coin", and the residual value will be 5 – 1 = 4

1 + cache[5 – 1] = 1 + cache [4] = 1 + 4 = 5

For 5 cents:

"I need one of 5 cents coin", and the residual value will be 5 – 5 = 0

1 + cache[5 – 5] = 1 + cache [0] = 1 + 0 = 1

optimal solution = Min (two solutions above) = Min (5, 1) = 1

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	1+1 = 2	1+2 = 3	1+3 = 4	Min(5,1) = 1		

Fill up num coins needed for a sum of 5 … the denominations that work are 1 and 5

For 1 cent:

"I need one of 1cent coin", and the residual value will be 5 – 1 = 4

1 + cache[5 – 1] = 1 + cache [4] = 1 + 4 = 5

For 5 cents:

"I need one of 5 cents coin", and the residual value will be 5 – 5 = 0

1 + cache[5 – 5] = 1 + cache [0] = 1 + 0 = 1

optimal solution = Min (two solutions above) = Min (5, 1) = 1

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	1+1 = 2	1+2 = 3	1+3 = 4	Min(5,1) = 1		

Fill up num coins needed for a sum of 6 … the denominations that work are 1 and 5

For 1 cent:

"I need one of 1cent coin", and the residual value will be 6 – 1 = 5

1 + cache[6 – 1] = 1 + cache [5] = 1 + 1 = 2

For 5 cents:

"I need one of 5 cents coin", and the residual value will be 6 – 5 = 1

1 + cache[6 – 5] = 1 + cache [1] = 1 + 1 = 2

optimal solution = Min (two solutions above) = Min (2, 2) = 2

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

Sum	0	1	2	3	4	5	6	7
Num coins	0	1	1+1 = 2	1+2 = 3	1+3 = 4	Min(5,1) = 1	Min(2,2) = 2	Min(1 + cache [7-1], 1 + cache[7-5]) Min (3, 3) = 3

Fill up num coins needed for a sum of 6 … the denominations that work are 1 and 5

For 1 cent:

"I need one of 1cent coin", and the residual value will be 6 – 1 = 5

1 + cache[6 – 1] = 1 + cache [5] = 1 + 1 = 2

For 5 cents:

"I need one of 5 cents coin", and the residual value will be 6 – 5 = 1

1 + cache[6 – 5] = 1 + cache [1] = 1 + 1 = 2

optimal solution = Min (two solutions above) = Min (2, 2) = 2

Change Making problem

Coin denomination given: { 1, 5, 10, 25 }

For a value of 10:

Note that value of 9 would be computed as : 5cents + (1cent X 4) which is a total of 5 coins.

Sum	0	1	2	3	4	5	6	7	9	10
Num coins	0	1	2	3	4	1	2	Min (3, 3) = 3	5	Min(1 + cache [10-1], 1 + cache[10-5], 1 + cache [10 – 10]) = Min (1+5, 1+1, 1+0) = 1

Change Making problem

Bottom-up pseudocode:

```
int GetMinCoins ( int N )
        cache[0] = 0;

        for  value = 1  to  N
                int minCoins =  MAX_INT

                foreach  coin
                        if   (coin > value)
                                continue;

                        residualValue  = value – coin;
                        minCoins  =    MIN (   1 + cache [ residualValue ],    minCoins  );

                cache [ value ] = minCoins;

return  cache[N];
```

← In the last iteration, we will write the result for cache [N]

Time complexity: [redacted]

Space complexity: [redacted]

Change Making problem

Bottom-up pseudocode:

```
int GetMinCoins ( int N )
        cache[0] = 0;

        for  value = 1  to  N
                int minCoins =  MAX_INT

                foreach  coin
                        if   (coin > value)
                                continue;

                        residualValue = value – coin
                        minCoins  =   MIN ( 1 + cache [ residualValue ],  minCoins );

                cache [ value ] = minCoins;

return  cache[N];
```

← In the last iteration, we will write the result for cache [N]

Time complexity: O (N*C) ← In non-DP approach, we saw exponential complexity

Space complexity: O (N)

Change Making – Summary - complexity

Approach	Time	Space
Naïve recursive	Exponential O (C^N)	O (N) due to recursion
Memoization	O(N * C)	O (N) due to cache (as well as due to recursion)
Tabulation	O(N * C)	O (N) due to cache

LAB: Change Making

Coin denomination (cents) given: { 1, 2, 5 }

Target sum N = 14 cents

Write a function that returns the minimum number of coins needed to make the target sum.

1. Using top-down approach:
   - Your function will look something like:
   - GetMinCoinsTopDown (N)

2. Using bottom-up approach:
   - Your function will look something like:
   - GetMinCoinsBottomUp (N)

Edit distance

This is a harder problem in DP.

Goal is to convert a source string to a destination string by changing the source string's characters.

Each such "change" is a step.

And the minimum number of steps for the conversion is the edit distance between the two strings.

Hamming vs Edit distance

Hamming distance (HD):

Number of positions in which two strings of equal length are different.

“abcd” and “abce” : HD is 1

“abc” and “abc” : HD is 0

“abcdef” and “bcdefg” : HD is 6, since all chars in corresponding positions are different, even though 5 chars are the same.

a b c d e f
b c d e f g

Edit distance

Hamming distance (HD):

Number of positions in which two strings of equal length are different.

Edit distance:

Minimum number of steps for the conversion of one string to another.

“abcd” and “abce”	: HD is 1.	Edit Distance is 1
“abc” and “abc”	: HD is 0.	Edit Distance is 0
“abcdef” and “bcdefg”	: HD is 6.	Edit Distance is 2 (delete ‘a’, insert ‘g’)

Edit distance

We have to convert string A to string B

A = “abcde”

B = “ghcpt”

Choices (to convert A to B) are:

- Replace a character (R)
- Delete a character (D)
- Insert a character (I)

Edit distance

We have to convert string A to string B

A = “abcde”
B = “ghcpt”

Choices (to convert A to B) are:

- Replace a character (R)
- Delete a character (D)
- Insert a character (I)

- Now, to convert A to B, we could do something naïve such as:
    1. First, delete all characters of A
    2. Then, insert all characters of B

    OR

    …

Edit distance

We have to convert string A to string B

A = “abcde”

B = “ghcpt”

Choices (to convert A to B) are:

- Replace a character (R)
- Delete a character (D)
- Insert a character (I)

- Now, to convert A to B, we could do something naïve such as:
  1. First, delete all characters of A
  2. Then, insert all characters of B

  OR

  We could figure out something efficient… if this is the path with least steps (where each step is either R or D or I), then that is the edit distance between the two strings.

Edit distance

We have to convert string A to string B

A = “abcde”
B = “ghcpt”

What options do we have at each step (i.e., at each character)?
EditD (A, B) =
EditD (“abcde”, “ghcpt”) = Min (
EditD (“abcd”, “ghcpt”) + 1, // *Delete* char from A, now need to convert remaining A to B
…

Edit distance

We have to convert string A to string B

A = “abcde”
B = “ghcpt”

What options do we have at each step (i.e., at each character)?
EditD (A, B) =
EditD (“abcde”, “ghcpt”) = Min (
EditD (“abcd”, “ghcpt”) + 1, // *Delete* char from A, now need to convert remaining A to B
EditD (“abcde**%**”, “ghcpt”) + 1, // *Insert* char (say, %) into A, now need to convert resulting A’ to B
…

Edit distance

We have to convert string A to string B

A = “abcde”
B = “ghcpt”

What options do we have at each step (i.e., at each character)?
EditD (A, B) =
EditD (“abcde”, “ghcpt”) = Min (

EditD (“abcd”, “ghcpt”) + 1,	// *Delete* char from A, now need to convert remaining A to B
EditD (“abcde**%**”, “ghcpt”) + 1,	// *Insert* char (say, %) into A, now need to convert resulting A’ to B
EditD (“abcd”, “ghcp”) + 1	// *Replace* character in A, now need to convert remaining A (“abcd”)
)	// to remaining B (“ghcp”)

We are assuming cost of any operation is 1 (hence we have + 1 in each option above).
We could generalize the cost (next)

Edit distance

We have to convert string A to string B

A = “abcde”
B = “ghcpt”

What options do we have at each step (i.e., at each character)?

Generalizing cost, we could write it as:

EditD (“abcde”, “ghcpt”) = Min (
EditD (“abcd”, “ghcpt”) + D_{cost}, // *Delete* char from A, now need to convert remaining A to B
EditD (“abcde**%**”, “ghcpt”) + I_{cost}, // *Insert* char into A, now need to convert resulting A’ to B
EditD (“abcd”, “ghcp”) + R_{cost} // *Replace* character in A, now need to convert remaining A (“abcd”)
) // to remaining B (“ghcp”)

For simplicity of our discussion (and more importantly, the equations☺), we will assume all costs as 1

Edit distance

We will use the following convention:

A [i] refers to the string A [0 .. i] ← NOTE: A [i] does NOT refer to just the i^{th} character

and [i, j] refers to the string A [0 .. i] and B [0 .. j]

EditD (“abcde”, “ghcpt”) = Min (

EditD (“abcd”, “ghcpt”) + 1, // *Delete* char from A, now need to convert remaining A to B

EditD (“abcde**%**”, “ghcpt”) + 1, // *Insert* char into A, now need to convert resulting A’ (A prime) to B

EditD (“abcd”, “ghcp”) + 1 // *Replace* character in A, now need to convert remaining A (“abcd”)

) // to remaining B (“ghcp”)

Given the convention, we will rewrite above as:

EditD (i, j) = Min (

EditD (i - 1, j) + 1, // *Delete* i^{th} char from A, now need to convert A [i - 1] to B [j]

…

Edit distance

Now, lets replace the strings in our equation with some indices.

We will use the following convention:

A [i] refers to the string A [0 .. i] ← NOTE: A [i] does NOT refer to just the i^{th} character

and [i, j] refers to the string A [0 .. i] and B [0 .. j]

Given above convention, we will rewrite:

```
EditD ( “abcde”, “ghcpt” ) =  Min (
                              EditD ( “abcd”, “ghcpt” )  + 1,      // Delete char from A, now need to convert remaining A to B
                              EditD ( “abcde%”, “ghcpt” )  + 1,    // Insert char into A, now need to convert resulting A’ (A prime) to B
                              EditD ( “abcd”, “ghcp” )  + 1        // Replace character in A, now need to convert remaining A (“abcd”)
                              )                                    // to remaining B ( “ghcp”)
```

as:

```
EditD ( i, j ) =  Min (
                  EditD ( i - 1,   j     ) + 1,          // Delete  ith char from A, now need to convert A [ i - 1 ]  to B [ j ]
                  EditD ( i,       j - 1 ) + 1,          // Insert char into A ➔ Last char of B taken care of, now convert A[0..i]  to  B[0..j-1]
                    …
```

Edit distance

Now, lets replace the strings in our equation with some indices.

We will use the following convention:

A [i] refers to the string A [0 .. i] ← NOTE: A [i] does NOT refer to just the i^{th} character

and [i, j] refers to the string A [0 .. i] and B [0 .. j]

Given above convention, we will rewrite:

```
EditD ( “abcde”, “ghcpt” ) =  Min (
                                EditD ( “abcd”, “ghcpt” )  + 1,      // Delete char from A, now need to convert remaining A to B
                                EditD ( “abcde%”, “ghcpt” )  + 1,    // Insert char into A, now need to convert resulting A’ (A prime) to B
                                EditD ( “abcd”, “ghcp” )  + 1        // Replace character in A, now need to convert remaining A (“abcd”)
                                )                                    // to remaining B ( “ghcp”)
```

as:

```
EditD ( i, j ) =  Min (
                    EditD ( i - 1,   j     ) + 1,          // Delete  ith char from A, now need to convert A [ i - 1 ]  to B [ j ]
                    EditD ( i,       j - 1 ) + 1,          // Insert char into A ➔ Last char of B taken care of, now convert A[0..i]  to  B[0..j-1]
                    EditD ( i - 1,  j - 1 ) + 1            // Replace ith char in A, now need to convert  A [ i - 1 ]  to B [ j - 1 ]
                  )
```

Next: recurrence relation

Edit distance

Now u have this as the recurrence relation:

TopDown ➔:

EditD (i, j) = Min (

EditD (i - 1, j) + 1, // *Delete* i^{th} char from A, now need to convert A [i - 1] to B [j]

EditD (i, j - 1) + 1, // Insert char into A ➔ Last char of B taken care of, now convert A[0..i] to B[0..j-1]

EditD (i - 1, j - 1) + 1) // *Replace* i^{th} char in A, now need to convert A [i - 1] to B [j - 1]

We will use the recurrence relation for solving it via DP.

Use above for top-down and next we will see for bottom-up

Edit distance

Now u have this as the recurrence relation:

Bottom-up ➔

```
cache [ i ][ j ] =  Min (  cache [  i - 1,    j  ]  + 1,
                           cache [  i,        j - 1 ]  + 1,
                           cache [  i - 1,   j - 1 ] + 1 )
```

OR

Taking the *+1* out since its common:

```
cache [ i ][ j ] =  1 +  Min (   cache [  i - 1,    j  ] ,
                                 cache [  i,        j - 1 ],
                                 cache [  i - 1,   j - 1 ])
```

Edit distance

Lets look at an example.

Given: Transform “ghc” to “abc”

Notice that the last character **is the same** (‘c’)
So, no work required for the last character, and this problem reduces to conversion of “gh” to “ab”

What that means is:

Edit distance (“ghc”, “abc”) is equal to Edit distance (“gh”, “ab”)

Edit distance

Given: Transform string A "ghcp" to string B "abcd"

Notice that the last character is **NOT the same** ('p' vs 'd')

To achieve this conversion, we will have multiple options, but eventually will take the min cost option:

1. Delete the last character 'p' from A

So,

A [3] ➔ B [3]

becomes:

A [2] ➔ B [3]

because we did a deletion in A, that did not affect B, so, the reduced problem is

A [2] ➔ B [3] (now, need to convert "ghc" to "abcd")

So,

Edit distance ("ghcp", "abcd") = Edit distance ("ghc", "abcd") + 1 // (1 is cost of deletion)

```
EditD ( i, j ) = Min (
                    EditD ( i - 1,  j    ) + 1,      // Delete  ith char from A, now need to convert A [ i - 1 ]  to B [ j ]
                    EditD ( i,      j - 1 ) + 1,     // Insert char into A ➔ Last char of B taken care of, now convert A[0..i]  to  B[0..j-1]
                    EditD ( i - 1,  j - 1 ) + 1 )    // Replace ith char in A, now need to convert  A [ i - 1 ]  to B [ j - 1 ]
```

Edit distance

Given: Transform string A "ghcp" to string B "abcd"

Notice that the last character is NOT the same ('p' vs 'd')

To achieve this conversion, we can:

2. Insert 'd' at end of B

So,

A [3] ➔ B [3]

becomes:

A [3] ➔ B [2]

because we did an insertion in B, it did not affect A, so, the reduced problem is

A [3] ➔ B [2] (now, need to convert "ghcp" to "abc")

So,

Edit distance ("ghcp", "abcd") = Edit distance ("ghcp", "abc") + 1 // (1 is cost of insertion)

```
EditD ( i, j ) = Min (
                    EditD ( i - 1,  j     ) + 1,      // Delete  ith char from A, now need to convert A [ i - 1 ]  to B [ j ]
                    EditD ( i,      j - 1 ) + 1,      // Insert char into A ➔ Last char of B taken care of, now convert A[0..i]  to  B[0..j-1]
                    EditD ( i - 1,  j - 1 ) + 1 )     // Replace ith char in A, now need to convert  A [ i - 1 ]  to B [ j - 1 ]
```

Edit distance

Given: Transform string A "ghcp" to string B "abcd"

Notice that the last character is NOT the same ('p' vs 'd')

To achieve this conversion, we can:

3. Replace 'p' with 'd'

So,

A [3] ➔ B [3]

becomes:

A [2] ➔ B [2]

because we replaced 'p' with a 'd' (which of course will have a cost of 1)

So,

Edit distance ("ghcp", "abcd") = Edit distance ("ghc", "abc") + 1 // (1 is cost of replacement)

EditD (i, j) = Min (

EditD (i - 1, j) + 1, // *Delete* i^{th} char from A, now need to convert A [i - 1] to B [j]

EditD (i, j - 1) + 1, // Insert char into A ➔ Last char of B taken care of, now convert A[0..i] to B[0..j-1]

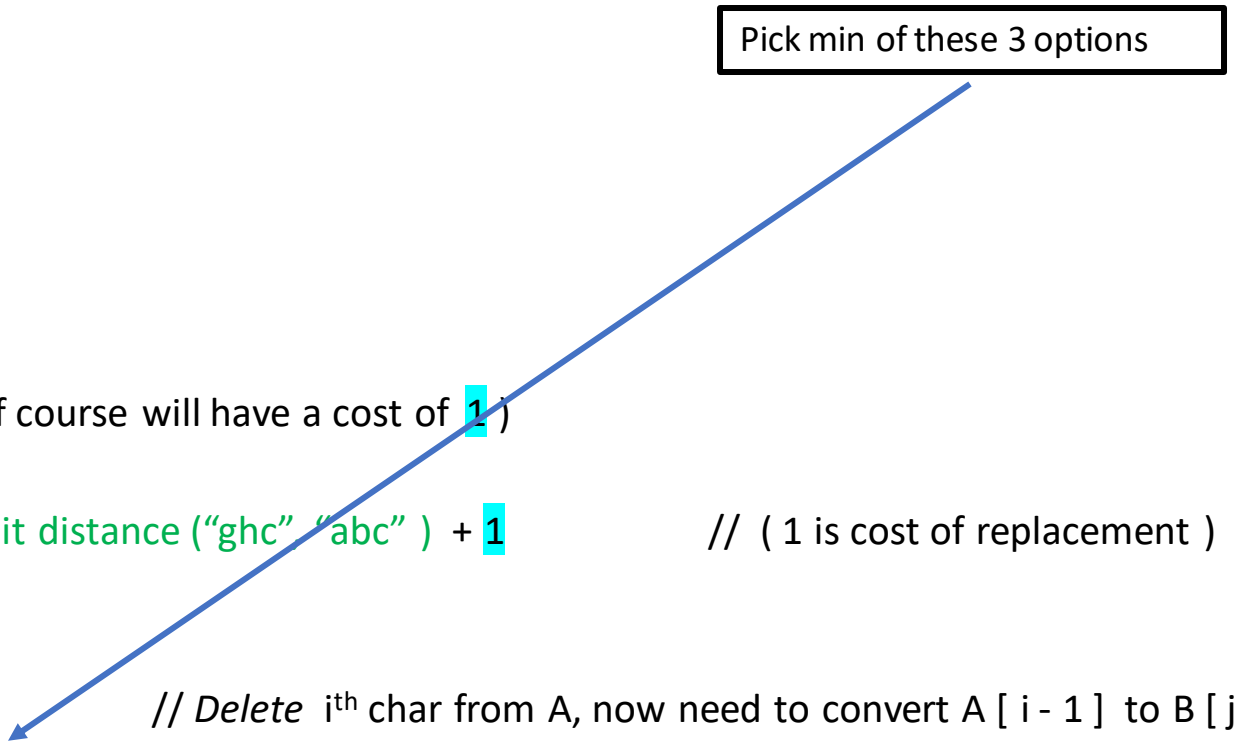
EditD (i - 1, j - 1) + 1) // *Replace* i^{th} char in A, now need to convert A [i - 1] to B [j - 1]

Edit distance

Given: Transform string A "ghcp" to string B "abcd"

Notice that the last character is NOT the same ('p' vs 'd')

To achieve this conversion, we can:



3. Replace 'p' with 'd'

So,

A [3] ➔ B [3]

becomes:

A [2] ➔ B [2]

because we replaced 'p' with a 'd' (which of course will have a cost of 1)

So,

Edit distance ("ghcp", "abcd") = Edit distance ("ghc", "abc") + 1 // (1 is cost of replacement)

```
EditD ( i, j ) = Min (
                    EditD ( i - 1,  j     ) + 1,      // Delete  ith char from A, now need to convert A [ i - 1 ]  to B [ j ]
                    EditD ( i,      j - 1 ) + 1,      // Insert char into A ➔ Last char of B taken care of, now convert A[0..i]  to  B[0..j-1]
                    EditD ( i - 1,  j - 1 ) + 1 )     // Replace ith char in A, now need to convert  A [ i - 1 ]  to B [ j - 1 ]
```

Edit distance - complexity

Naïve recursive (top down):

Time: $O (3^{\min (M, N)})$

Space: $O (\min (M, N))$

Memoization (top down):

Time: $O (M \times N)$

Space: $O (M \times N)$

Tabulation (bottom up):

Time: $O (M \times N)$

Space: $O (M \times N)$

Edit distance

- Edit distance between “ghcpt” and “abcde” is 4 because we do all replaces except for ‘c’ which is common.
  - See MORE examples in code in IDE

	“”	g	h	c	p	t
“”	0	1	2	3	4	5
a	1	1	2	3	4	5
b	2	2	2	3	4	5
c	3	3	3	2	3	4
d	4	4	4	3	3	4
e	5	5	5	4	4	4