CSCI 232:
Data Structures and Algorithms

Dynamic Programming (Part 1)

Reese Pearsall

Summer 2025

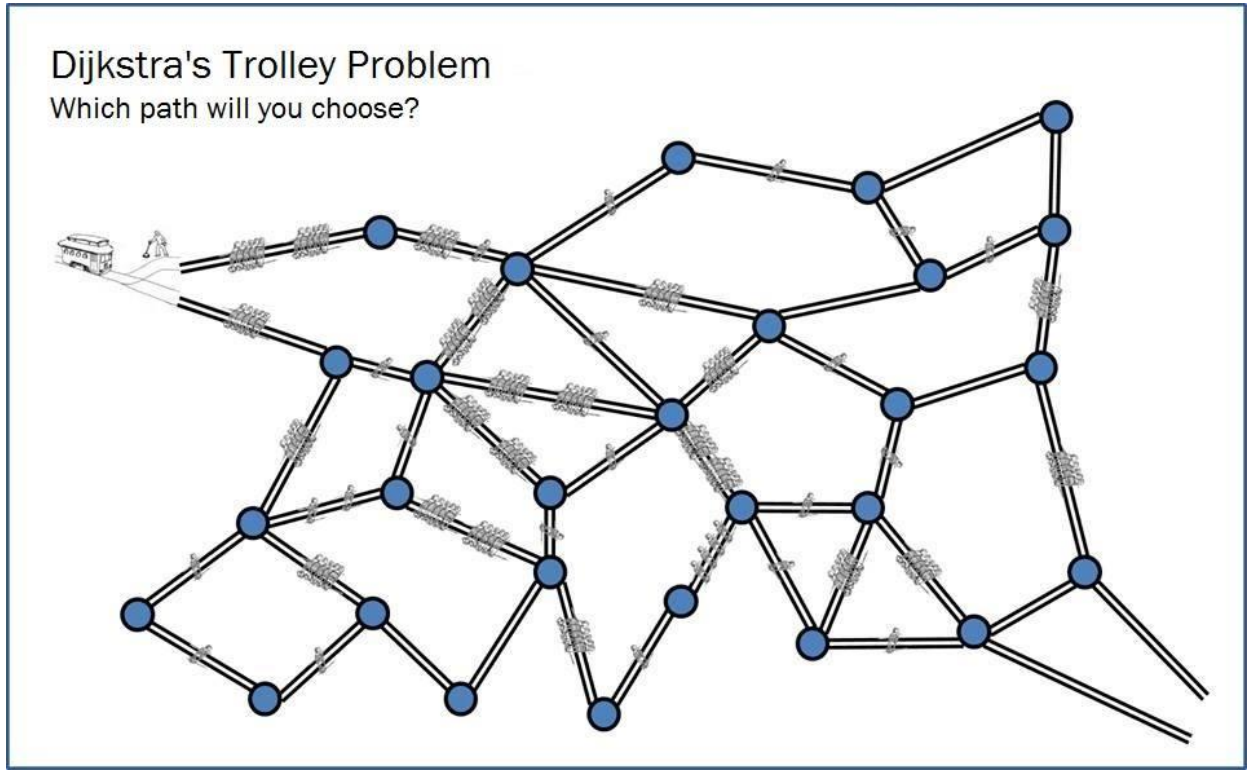
Announcements

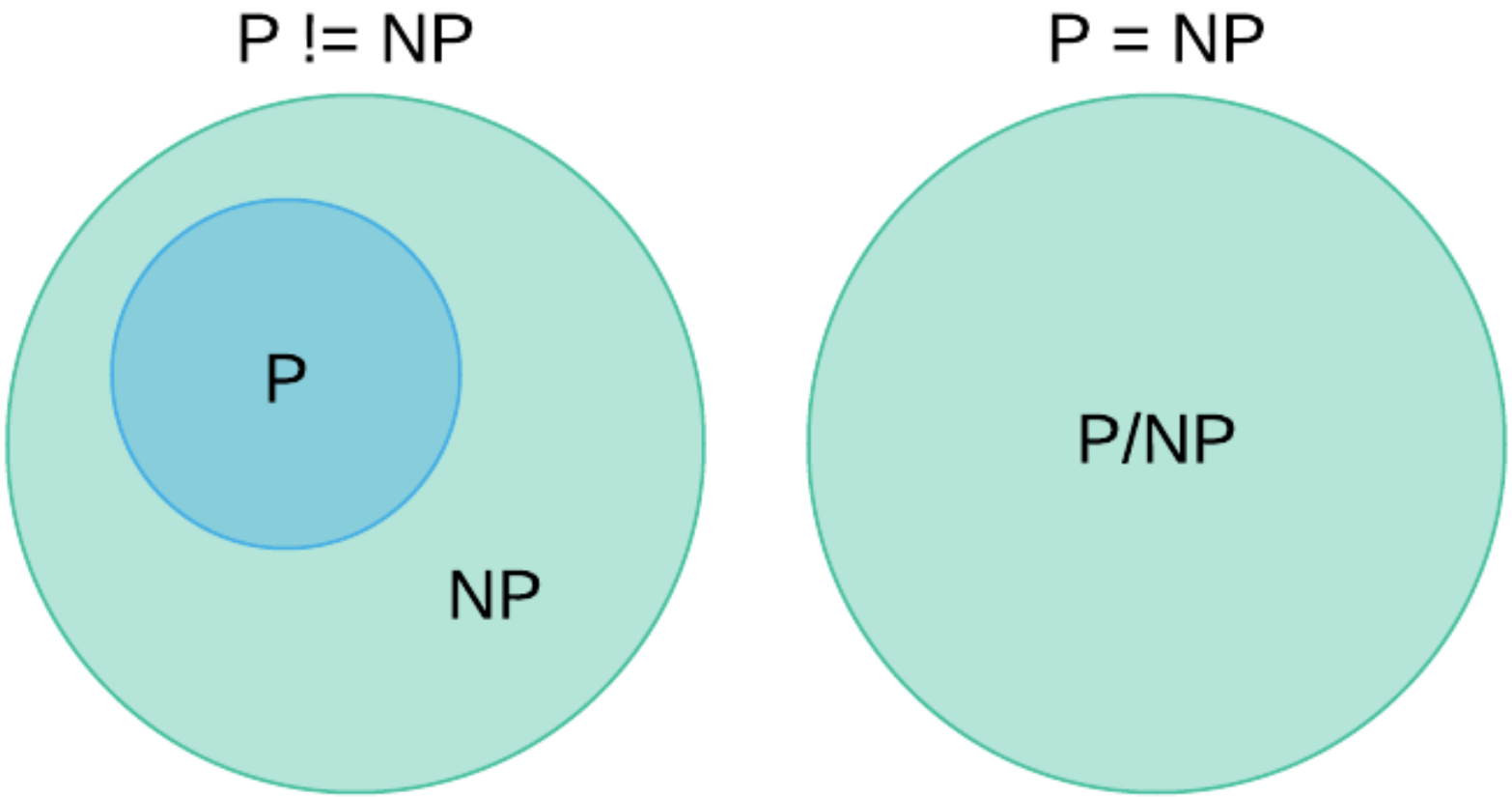
Program 3 Due tonight!!

Thursday will be Quiz day

Might be slightly late tomorrow

Last week of class!!




P != NP
P
NP
P = NP
P/NP

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 25]

K = 37

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 25]

K = 37

Answer = 4

(Quarter, dime, two pennies)

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 25]

K = 37

Answer = 4

(Quarter, dime, two pennies)

Algorithm?

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 25]

K = 37

Answer = 4

(Quarter, dime, two pennies)

Use as many quarters as possible, then as many dimes as possible, …

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 25]

K = 37

Answer = 4

(Quarter, dime, two pennies)

Use as many quarters as possible, then as many dimes as possible, …

This is known as the **greedy** approach

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 25]

K = 37

Greedy Algorithm

Use as many quarters as possible, then as many dimes as possible, …

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 18, 25]

K = 37

Greedy Algorithm

Use as many quarters as possible, then as many 18 cent pieces as possible, then dimes , …

What if there were also an 18-cent coin?

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 18, 25]

K = 37

Greedy Algorithm

Use as many quarters as possible, then as many 18 cent pieces as possible, then dimes , …

25, 10, 1, 1 **(4 coins)**

What if there were also an 18-cent coin?

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 18, 25]

K = 37

Greedy Algorithm

Use as many quarters as possible, then as many 18 cent pieces as possible, then dimes , …

25, 10, 1, 1 (4 coins)

What if there were also an 18-cent coin?

Real Answer = 18, 18, 1 (3 coins)

Change Making Problem

Given a set a coin denominations **D**, how can you represent **K** cents with the smallest number of coins?

D = [1, 5, 10, 18, 25]

K = 37

Greedy Algorithm

Use as many quarters as possible, then as many 18 cent pieces as possible, then dimes , …

25, 10, 1, 1 (4 coins)

What if there were also an 18-cent coin?

Real Answer = 18, 18, 1 (3 coins)

Lesson Learned: The Greedy approach works for the United States denominations, but not for a general set of denominations

Change Making Problem

Suppose I tell you that 2 quarters, 1 dime, and 3 pennies are the minimum number of coins needed to make **63 cents**

(We will assume we have the standard US denominations [1, 5, 10, 25] (NO 50 CENT PIECE)

25 + 25 + 10 + 1 + 1 + 1 = 63

What can you conclude?

Does this provide an answer to any other change making problems?

Change Making Problem

Suppose I tell you that 2 quarters, 1 dime, and 3 pennies are the minimum number of coins needed to make **63 cents**

(We will assume we have the standard US denominations [1, 5, 10, 25] (NO 50 CENT PIECE)

25 + 25 + 10 + 1 + 1 + 1 = 63

This is the minimum coins needed to make 38 cents

Change Making Problem

Suppose I tell you that 2 quarters, 1 dime, and 3 pennies are the minimum number of coins needed to make **63 cents**

(We will assume we have the standard US denominations [1, 5, 10, 25] (NO 50 CENT PIECE)

25 + 25 + 10 + 1 + 1 + 1 = 63

This is the minimum coins needed to make 13 cents

Change Making Problem

Suppose I tell you that 2 quarters, 1 dime, and 3 pennies are the minimum number of coins needed to make **63 cents**

(We will assume we have the standard US denominations [1, 5, 10, 25] (NO 50 CENT PIECE)

25 + 25 + 10 + 1 + 1 + 1 = 63

This is the minimum coins needed to make 3 cents

Change Making Problem

Suppose I tell you that 2 quarters, 1 dime, and 3 pennies are the minimum number of coins needed to make **63 cents**

(We will assume we have the standard US denominations [1, 5, 10, 25] (NO 50 CENT PIECE)

25 + 25 + 10 + 1 + 1 + 1 = 63

This is the minimum coins needed to make 2 cents

Change Making Problem

Suppose I tell you that 2 quarters, 1 dime, and 3 pennies are the minimum number of coins needed to make **63 cents**

(We will assume we have the standard US denominations [1, 5, 10, 25] (NO 50 CENT PIECE)

25 + 25 + 10 + 1 + 1 + 1 = 63

This is the minimum coins needed to make 1 cent

Change Making Problem

Suppose I tell you that 2 quarters, 1 dime, and 3 pennies are the minimum number of coins needed to make **63 cents**

(We will assume we have the standard US denominations [1, 5, 10, 25] (NO 50 CENT PIECE)

25 + 25 + 10 + 1 + 1 + 1 = 63

The solution to the change making problems consists of solutions to smaller change making problems

We can use **recursion** to solve this problem

Change Making Problem

In general, suppose a country has coins with denominations:

$$1 = d_1 < d_2 < \cdots < d_k$$ (US coins: $d_1 = 1, d_2 = 5, d_3 = 10, d_4 = 25$)

Algorithm: To make change for p cents, we are going to figure out change for every value $x < p$. We will build solution for p out of smaller solutions.

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12)

We used one quarter

Now find the minimum number of coins needed to make 12 cents

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12)

We used one dime

C(12) = 1 + C(2)

Now find the minimum number of coins needed to make 2 cents

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12) 

C(12) = 1 + C(2) 

C(2) = 1 + C(1) 

C(1) = 1 + C(0) 

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12) 

C(12) = 1 + C(2) 

C(2) = 1 + C(1) 

C(1) = 1

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12) 

C(12) = 1 + C(2) 

C(2) = 1 + 1

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12) 

C(12) = 1 + C(2) 

C(2) = 2

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12) 

C(12) = 1 + 2 

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + C(12) 

C(12) = 3 

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 1 + 3

Change Making Problem

$C(p)$ – minimum number of coins to make p cents.

x – value (e.g. \$0.25) of a coin used in the optimal solution.

$$C(p) = 1 + C(p - x).$$

C(37) = 4

The minimum number of coins needed to make 37 cents is 4

Change Making Problem

In general, suppose a country has coins with denominations:

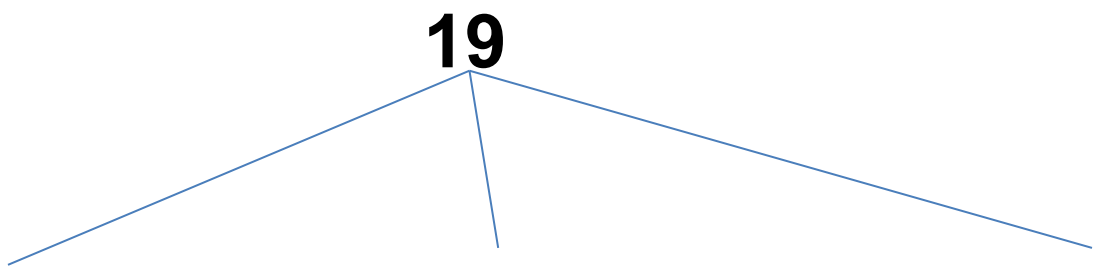
$$1 = d_1 < d_2 < \cdots < d_k$$ (US coins: $d_1 = 1, d_2 = 5, d_3 = 10, d_4 = 25$)

(This algorithm must work for ALL denominations)

Algorithm: To make change for p cents, we are going to figure out change for every value $x < p$. We will build solution for p out of smaller solutions.

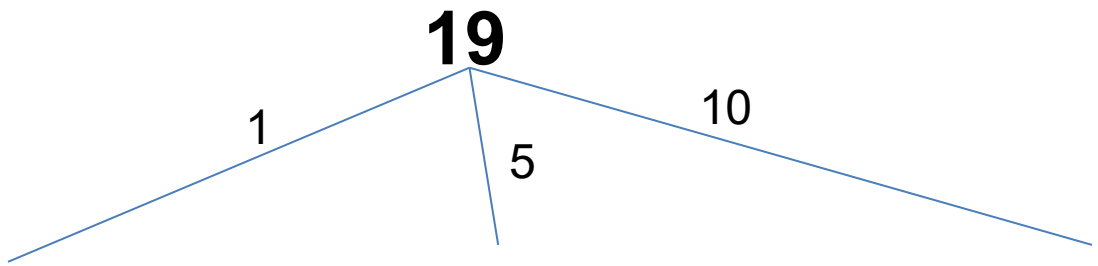
Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10



Change Making Problem

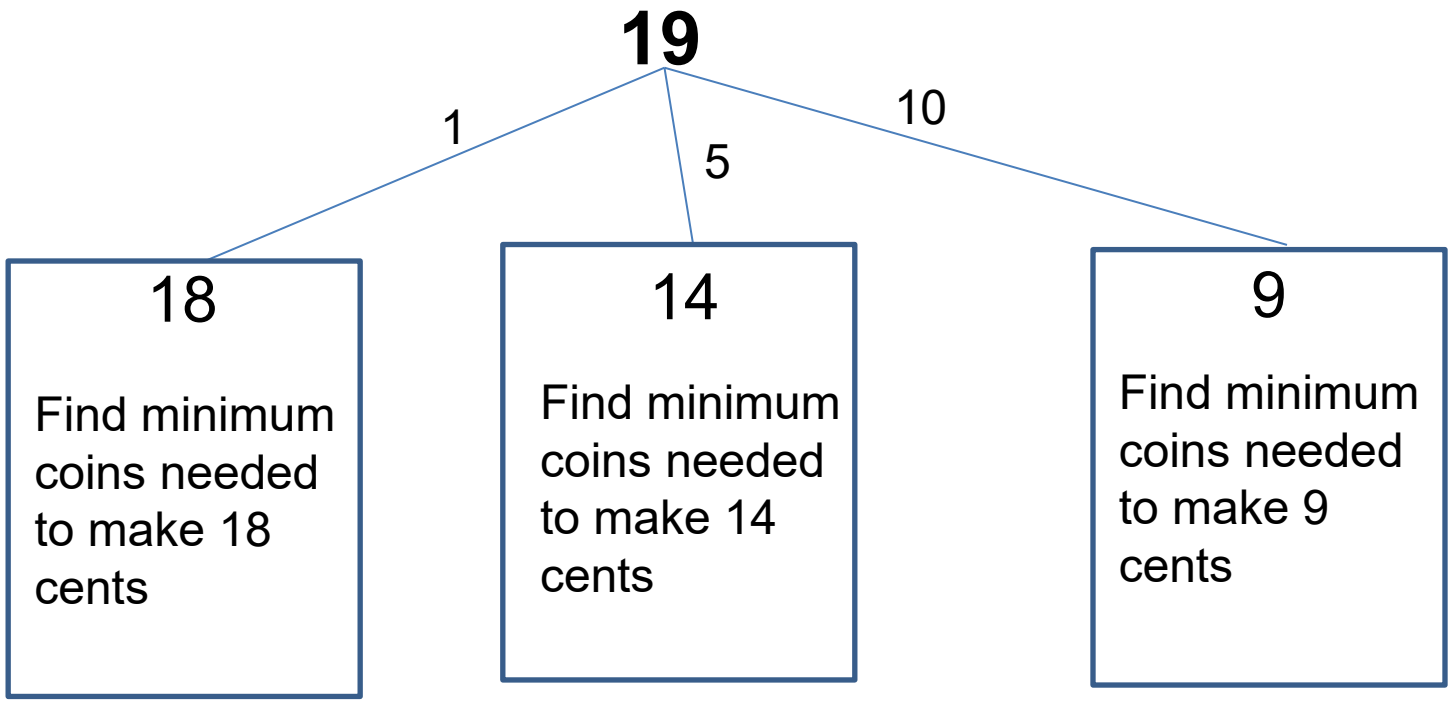
Make $0.19 with $0.01, $0.05, $0.10



Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

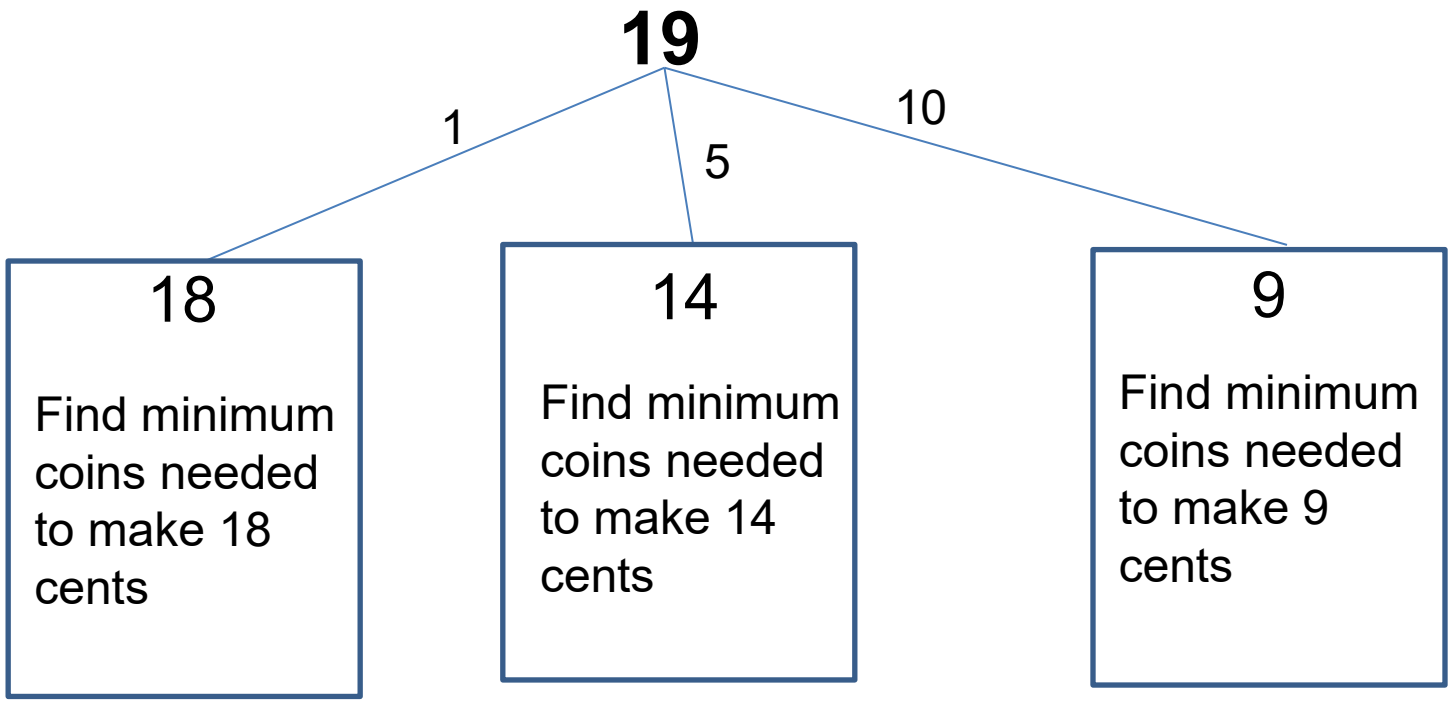


To find the minimum number of coins needed to create 19 cents, we generate **k** subproblems

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

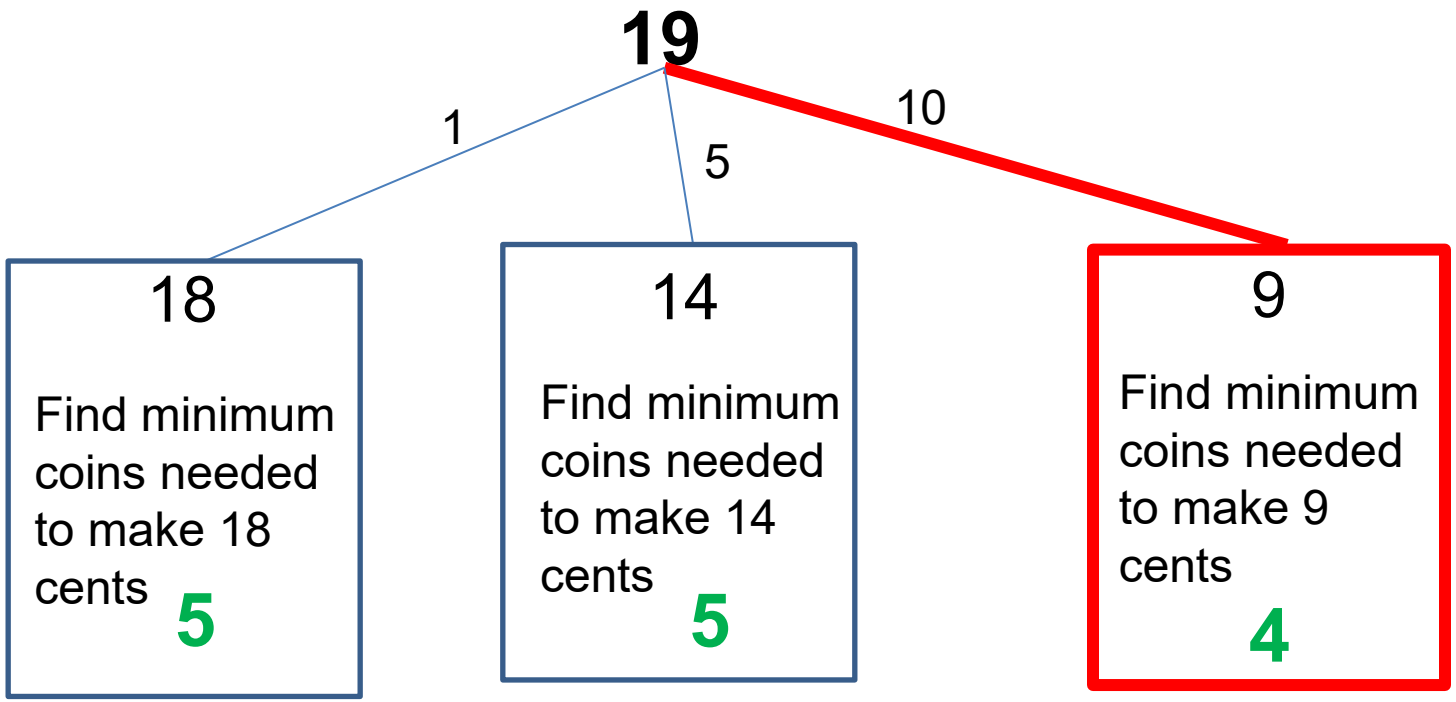


We want to select the **minimum** solution of these three subproblems

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

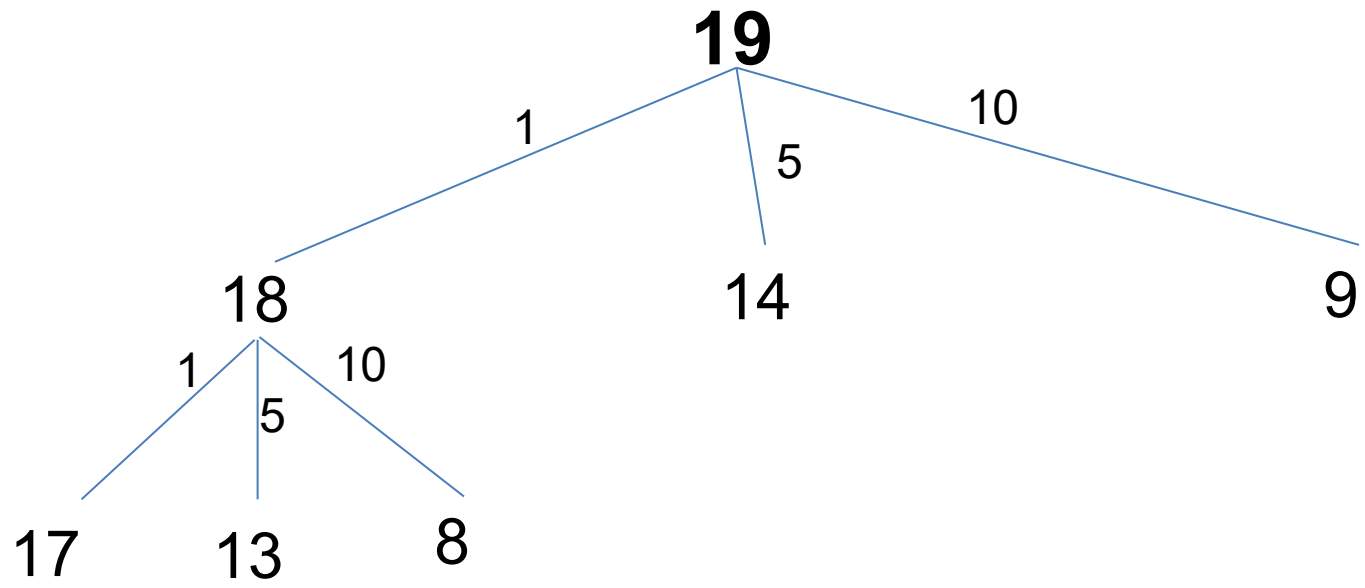


For the solution of our original problem (19), we want to select this branch (one dime used)

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations



Find minimum coins needed to make 17 cents

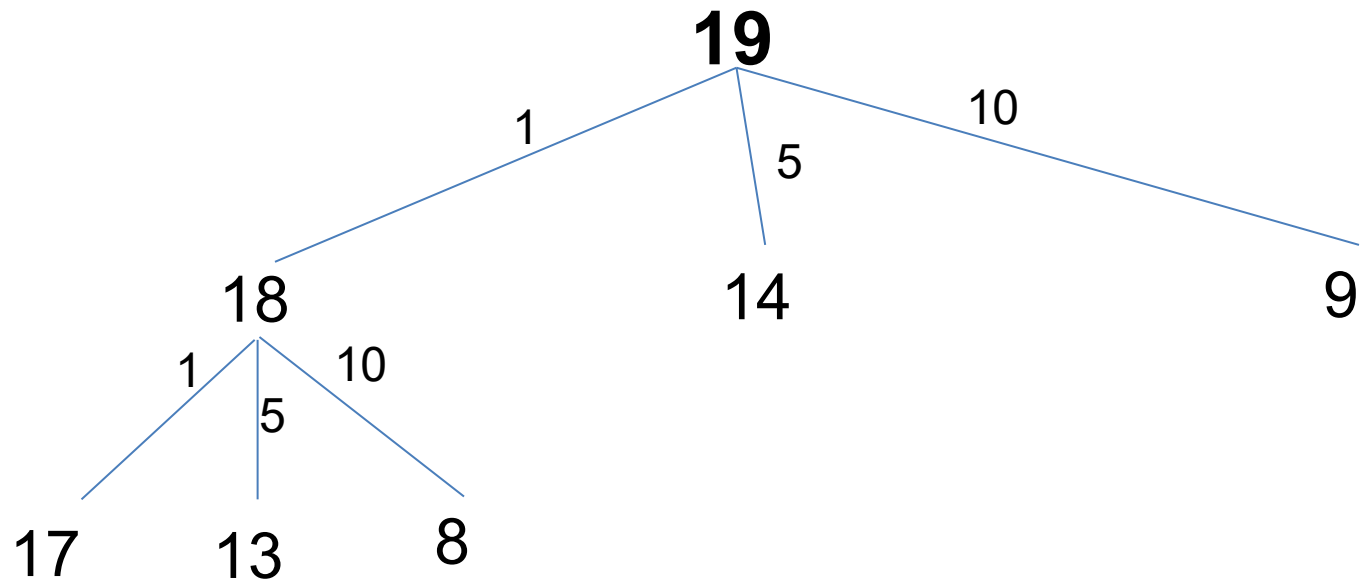
Find minimum coins needed to make 13 cents

Find minimum coins needed to make 8 cents

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

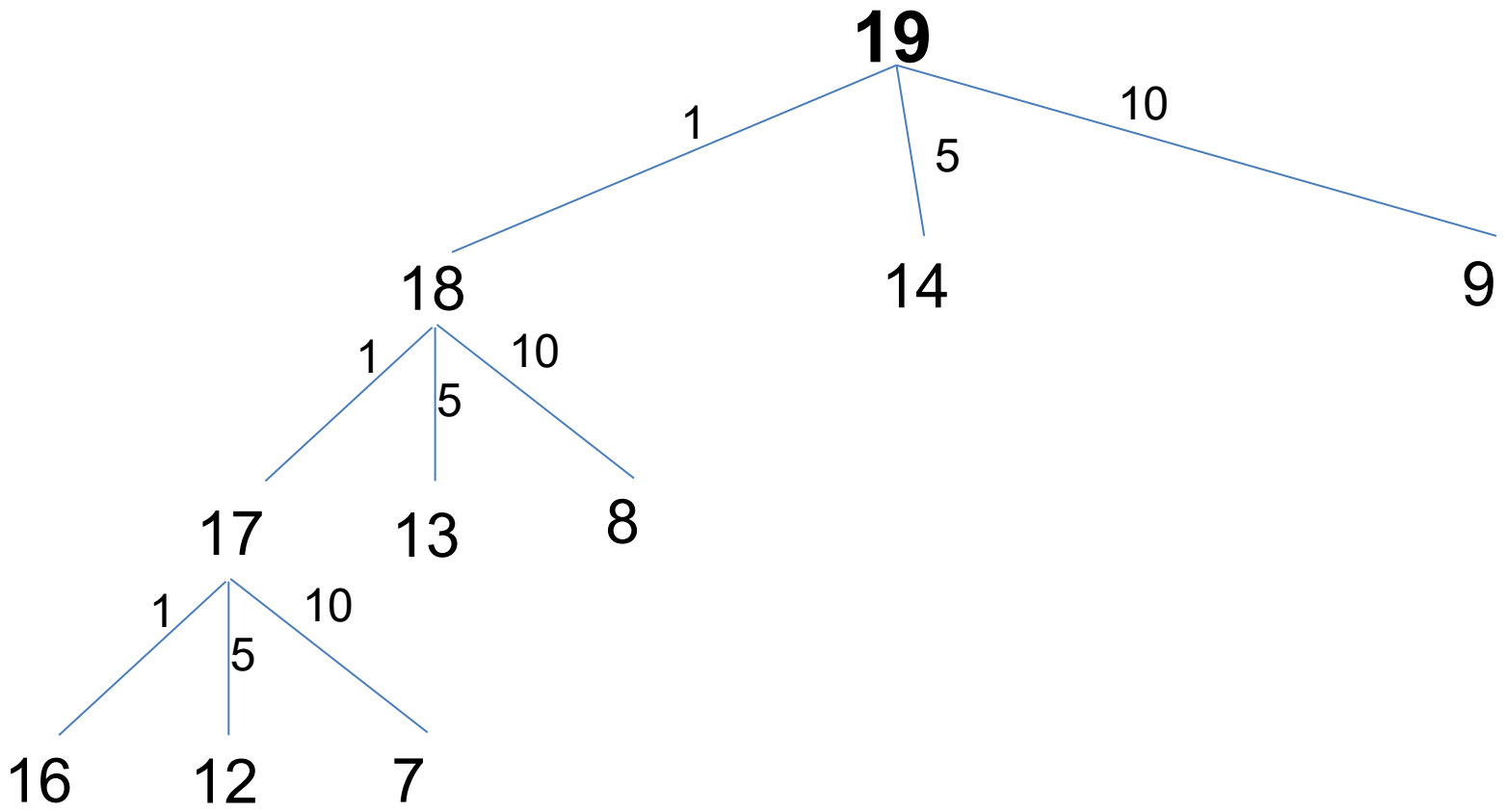
k = # denominations



Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

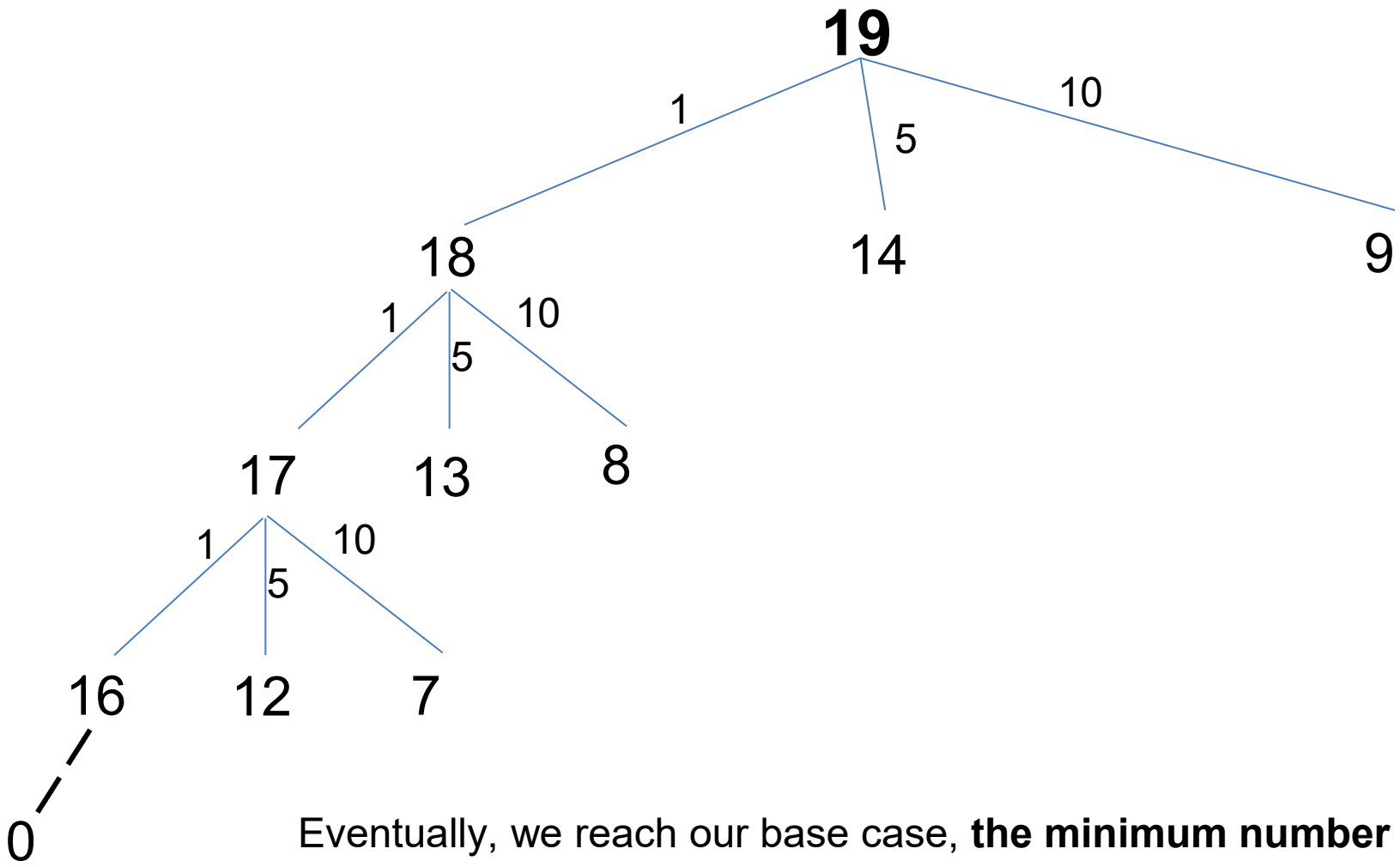
k = # denominations



Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

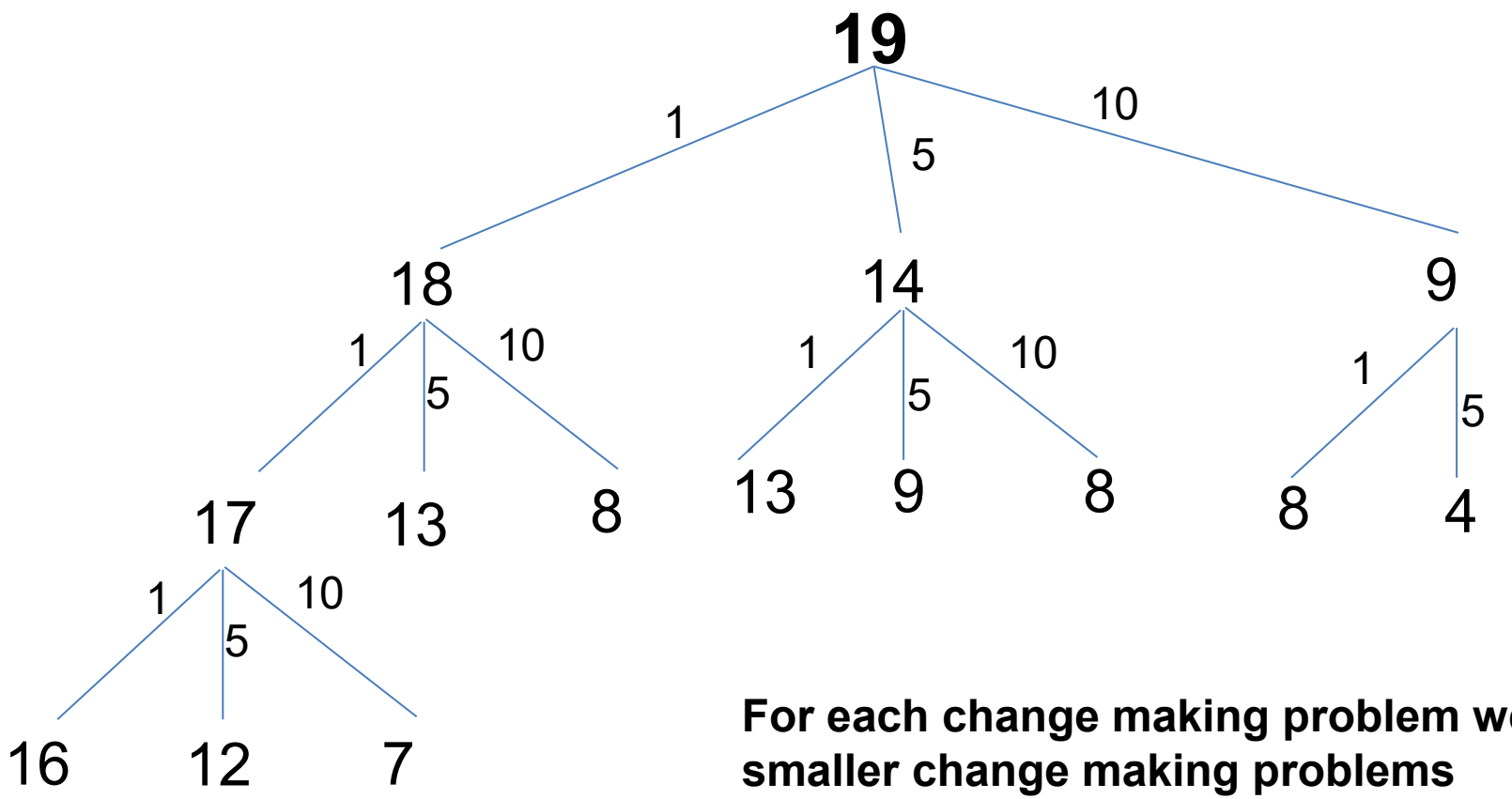


Eventually, we reach our base case, **the minimum number of coins needed to make 0 cents**

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations



When C(9), we cant use a 10 cent piece…

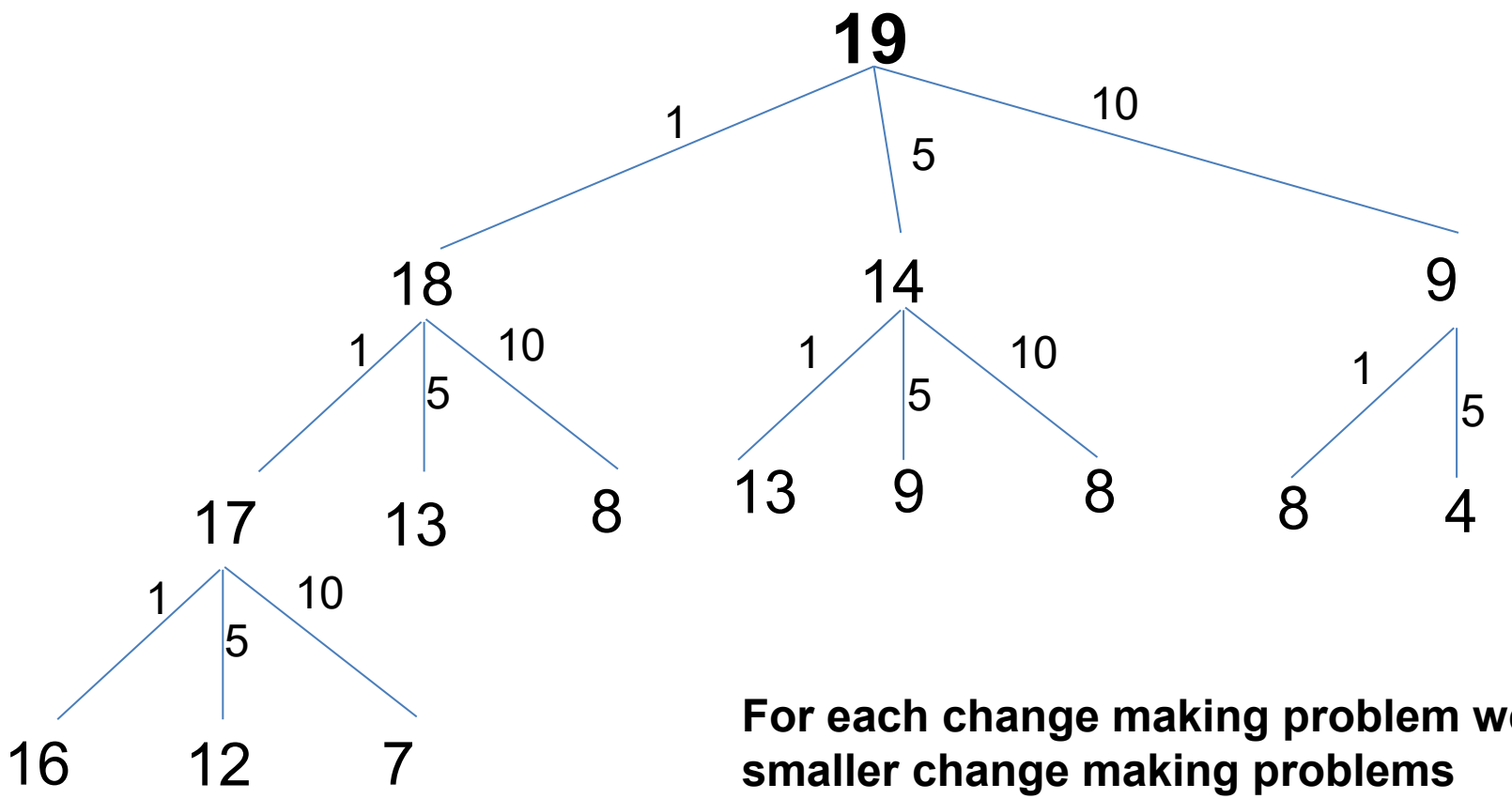
For each change making problem we solve, we must solve at most 3 smaller change making problems

Once we solve the smaller problems, we must select the branch that has the minimum value

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations



When C(9), we cant use a 10 cent piece…

For each change making problem we solve, we must solve at most 3 smaller change making problems

Once we solve the smaller problems, we must select the branch that has the minimum value

Change Making Problem

$$C(p) = \begin{cases} \min_{i:d_i \leq p} C(p - d_i) + 1, p > 0 \\ 0, p = 0 \end{cases}$$

Least change for 19 cents = minimum of:

- least change for 19-10 = 9 cents
- least change for 19-5 = 14 cents
- least change for 19-1 = 18 cents

For each problem P, we will solve the problem for (P – d), where d represents each possible denomination

Change Making Problem

$$C(p) = \begin{cases} \min_{i:d_i \leq p} C(p - d_i) + 1, p > 0 \\ 0, p = 0 \end{cases}$$

Least change for 19 cents = minimum of:

- least change for 19-10 = 9 cents
- least change for 19-5 = 14 cents
- least change for 19-1 = 18 cents

For each problem P, we will solve the problem for (P – d), where d represents each possible denomination

We want to select only the branch the yields the minimum value

Change Making Problem

$$C(p) = \begin{cases} \min_{i:d_i \leq p} C(p - d_i) + 1, p > 0 \\ 0, p = 0 \end{cases}$$

Least change for 19 cents = minimum of:

- least change for 19-10 = 9 cents
- least change for 19-5 = 14 cents
- least change for 19-1 = 18 cents

For each problem P, we will solve the problem for (P – d), where d represents each possible denomination

If we ever need to make change for 0 cents, return 0

We want to select only the branch the yields the minimum value

Change Making Problem

$$C(p) = \begin{cases} \min_{i:d_i \leq p} C(p - d_i) + 1, p > 0 \\ 0, p = 0 \end{cases}$$

Least change for 19 cents = minimum of:

- least change for 19-10 = 9 cents
- least change for 19-5 = 14 cents
- least change for 19-1 = 18 cents

For each problem P, we will solve the problem for (P – d), where d represents each possible denomination

If we ever need to make change for 0 cents, return 0

We want to select only the branch the yields the minimum value

Change Making Problem

D = array of denominations [1, 5, 10, 18, 25]
p = desired change (37)

```
min_coins(D, p)
```

Change Making Problem

D = array of denominations [1, 5, 10, 18, 25]
p = desired change (37)

```
min_coins(D, p)

    if p == 0
        return 0;
```

Base Case

Change Making Problem

D = array of denominations [1, 5, 10, 18, 25]
p = desired change (37)

```
min_coins(D, p)

    if p == 0
        return 0;

    else
        min = ∞
        a = ∞
```

Base Case

```
int min = Integer.MAX_VALUE;
int a = Integer.MAX_VALUE;;
```

Change Making Problem

D = array of denominations [1, 5, 10, 18, 25]
p = desired change (37)

```
min_coins(D, p)

    if p == 0
        return 0;

    else
        min = ∞
        a = ∞

        for each d_i in D
            if (p - d_i) >= 0
                a = min_coins(D, p - d_i )
```

Base Case

```
int min = Integer.MAX_VALUE;
int a = Integer.MAX_VALUE;;
```

Recurse, and find the minimum number of coins needed using each valid denomination

Change Making Problem

D = array of denominations [1, 5, 10, 18, 25]
p = desired change (37)

```
min_coins(D, p)

    if p == 0
        return 0;

    else
        min = ∞
        a = ∞

        for each d_i in D
            if (p - d_i) >= 0
                a = min_coins(D, p - d_i )
            if a < min
                min = a
```

Base Case

```
int min = Integer.MAX_VALUE;
int a = Integer.MAX_VALUE;;
```

Recurse, and find the minimum number of coins needed using each valid denomination

Select the branch that has the minimum value

Change Making Problem

D = array of denominations [1, 5, 10, 18, 25]
p = desired change (37)

```
min_coins(D, p)
    if p == 0
        return 0;
    else
        min = ∞
        a = ∞

        for each d_i in D
            if (p - d_i) >= 0
                a = min_coins(D, p - d_i )
            if a < min
                min = a
```

Base Case

```
int min = Integer.MAX_VALUE;
int a = Integer.MAX_VALUE;;
```

Recurse, and find the minimum number of coins needed using each valid denomination

Select the branch that has the minimum value

Change Making Problem

D = array of denominations [1, 5, 10, 18, 25]
p = desired change (37)

```
min_coins(D, p)
    if p == 0
        return 0;
    else
        min = ∞
        a = ∞

        for each d_i in D
            if (p − d_i) >= 0
                a = min_coins(D, p − d_i )
            if a < min
                min = a

        return 1 + min
```

Base Case

```
int min = Integer.MAX_VALUE;
int a = Integer.MAX_VALUE;;
```

Recurse, and find the minimum number of coins needed using each valid denomination

Select the branch that has the minimum value

Once, our for loop finishes, we should know the branch that had the minimum, so return (1 + min), 1 because one coin was used in the current method call

Change Making Problem

```
min_coins(D, p)
    if p == 0
        return 0;
    else
        min = ∞
        a = ∞

        for each d_i in D
            if (p - d_i) >= 0
                a = min_coins(D, p - d_i )
            if a < min
                min = a

      return 1 + min
```

Change Making Problem

```
min_coins(D, p)
    if p == 0
        return 0;
    else
        min = ∞
        a = ∞

        for each d_i in D
            if (p - d_i) >= 0
                a = min_coins(D, p - d_i )
            if a < min
                min = a

        return 1 + min
```

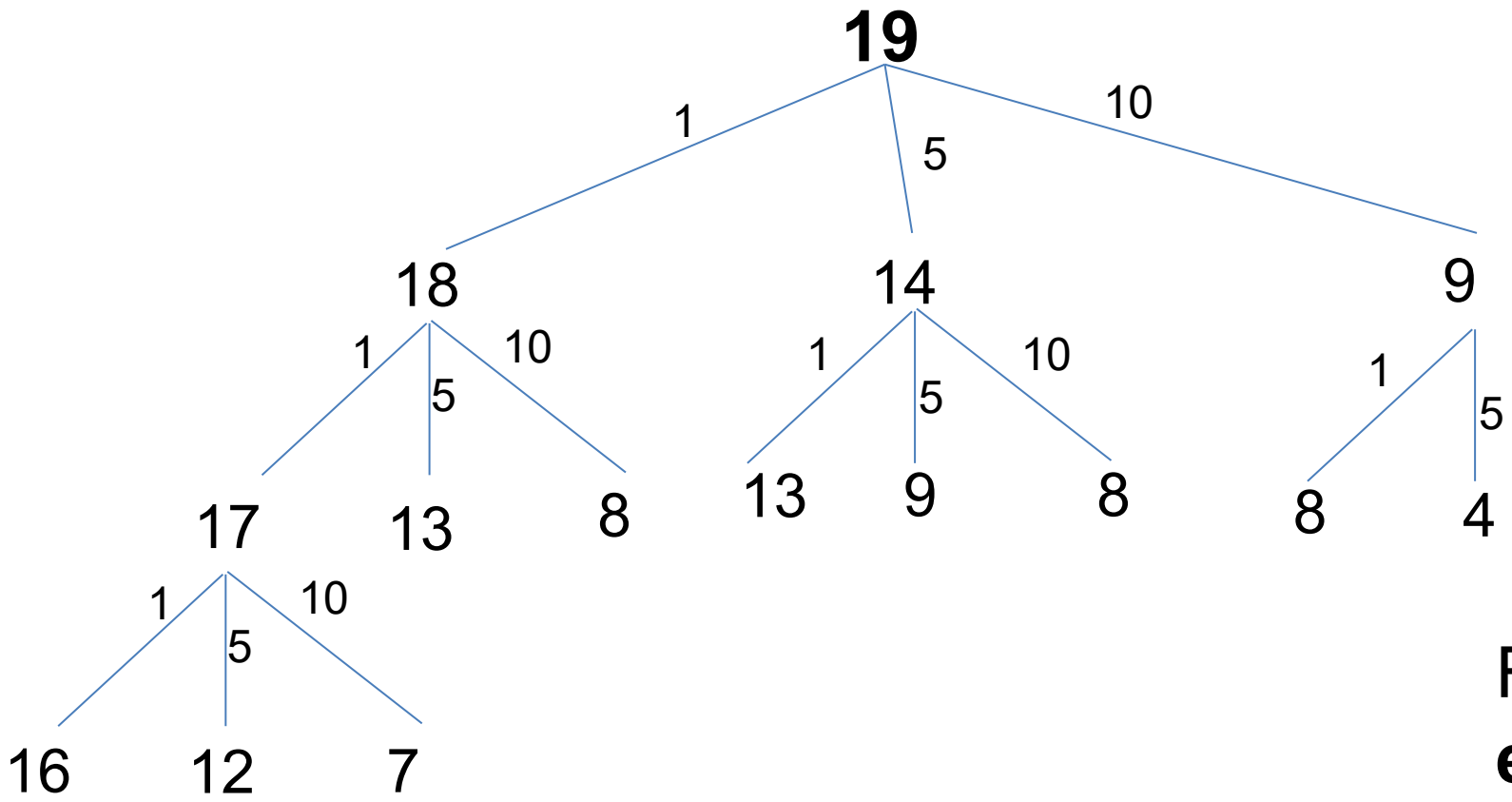
Running time?

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

p = value to make change for



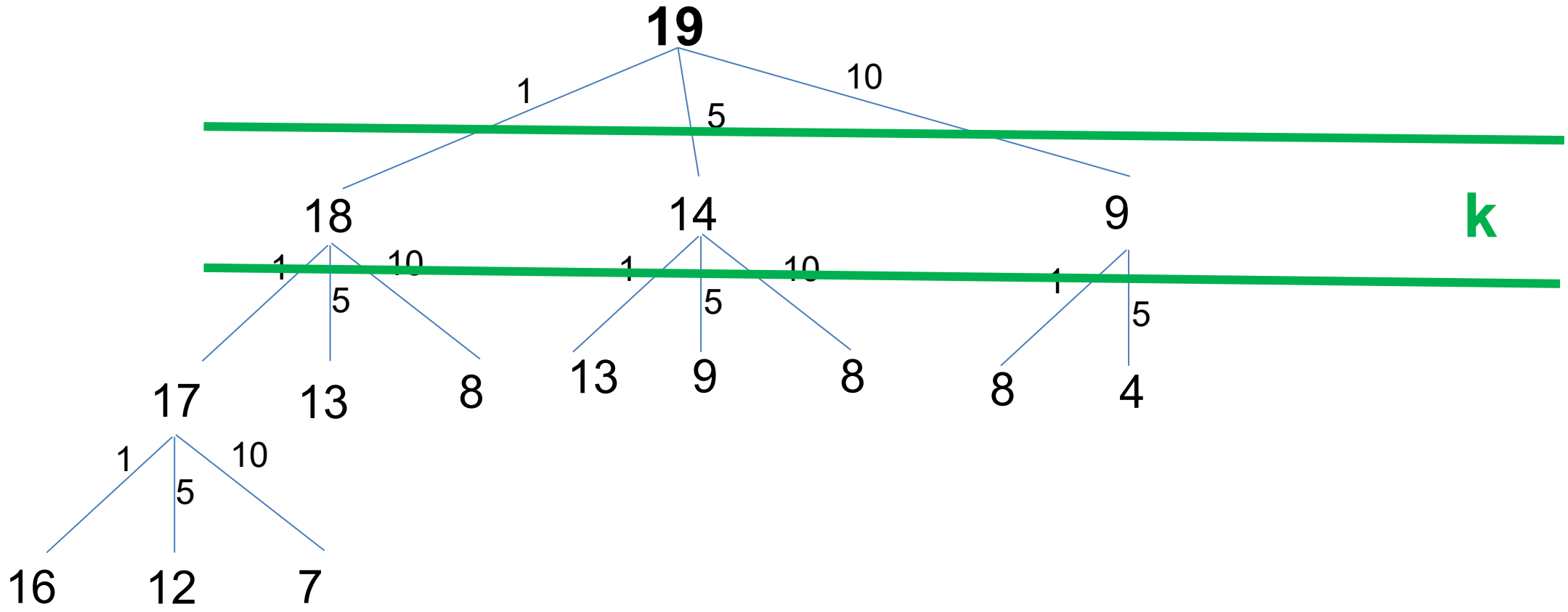
For sufficiently large p, **every permutation** of denominations is included.

Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

p = value to make change for

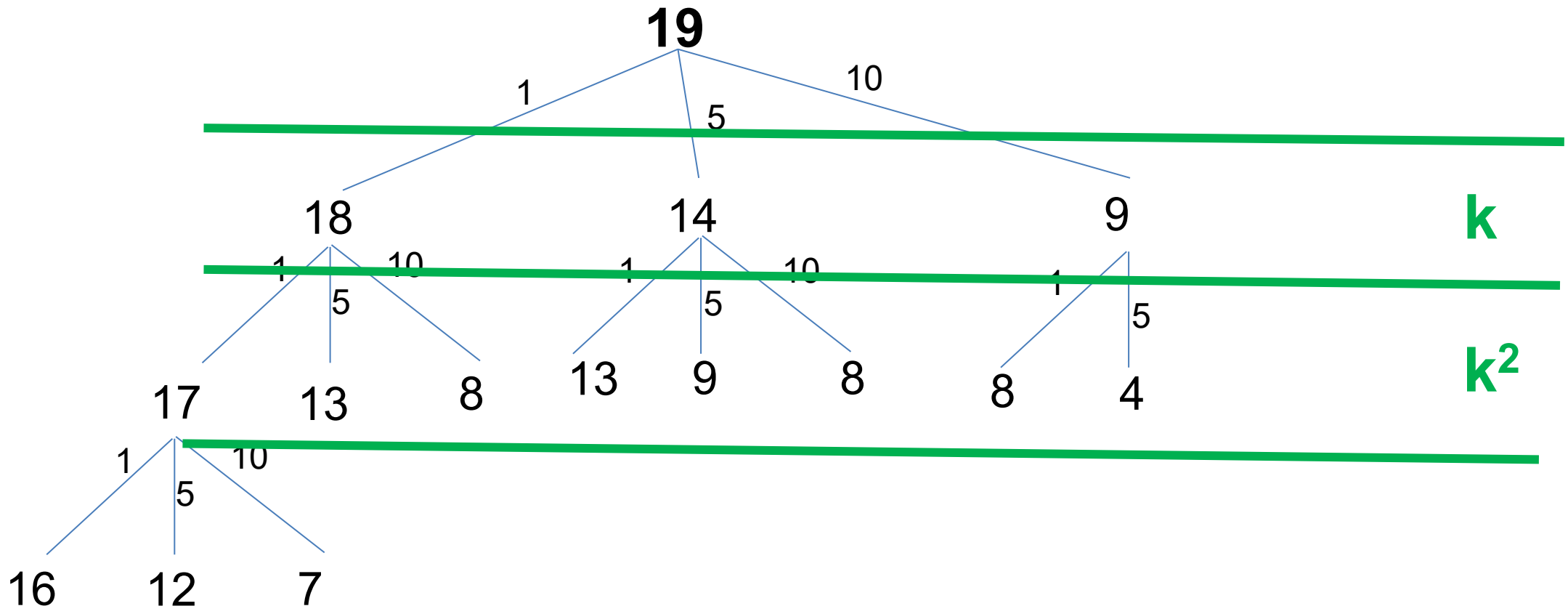


Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

p = value to make change for

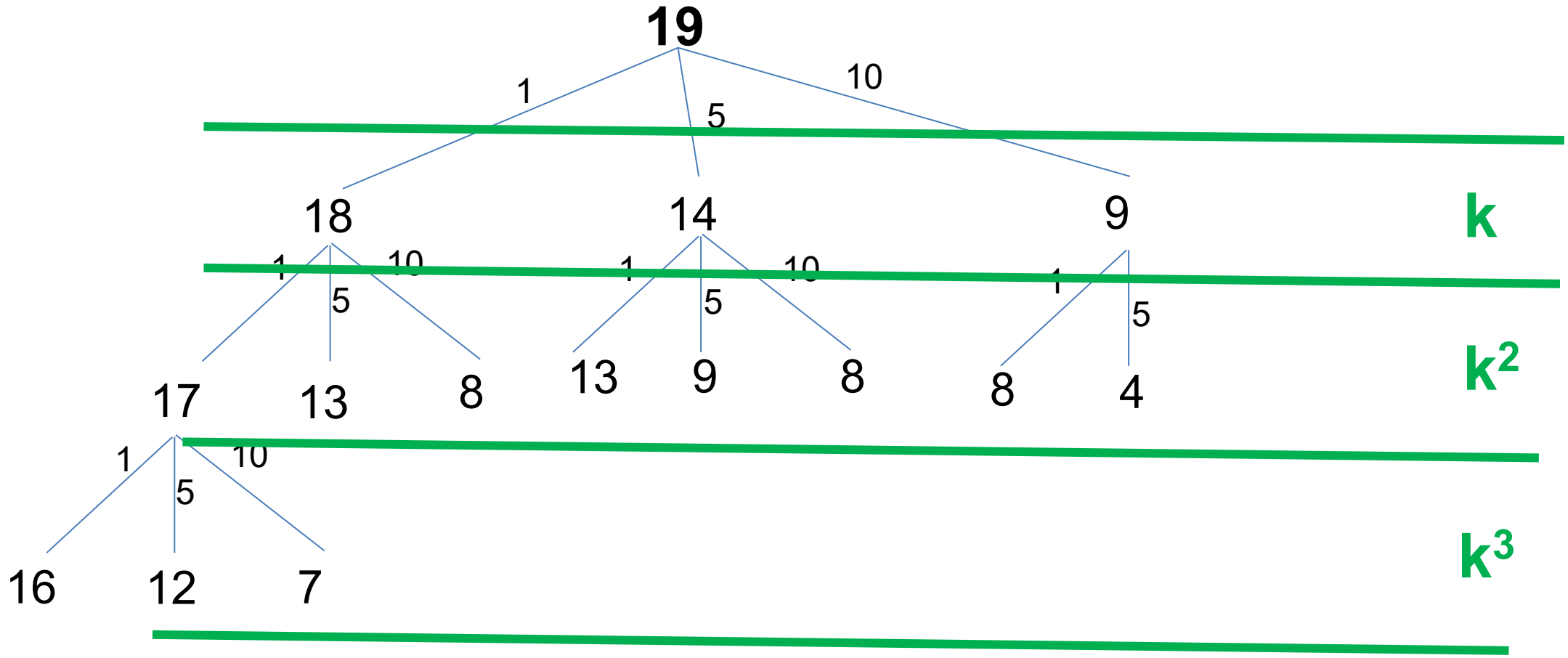


Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

p = value to make change for

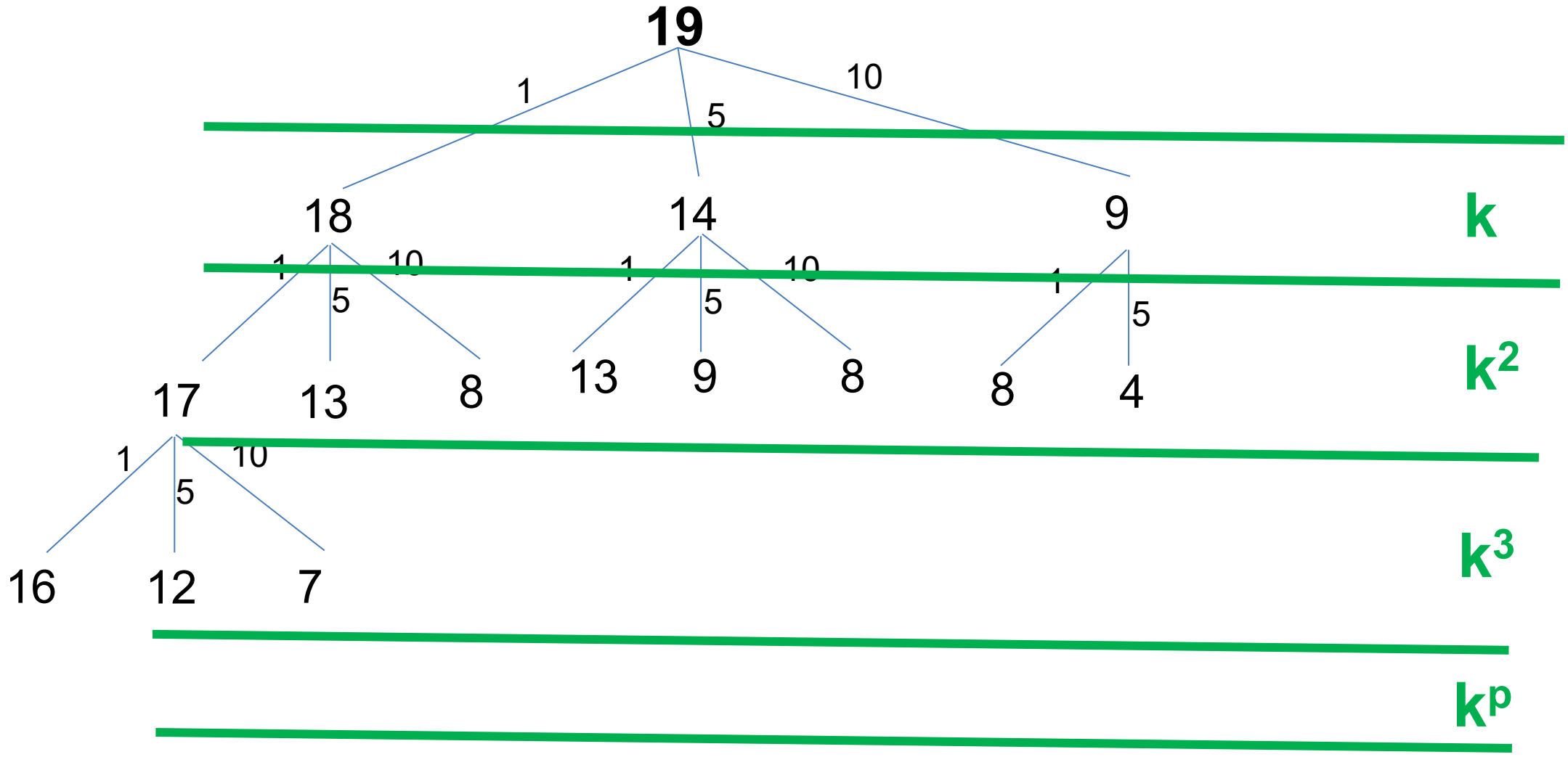


Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

p = value to make change for



Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

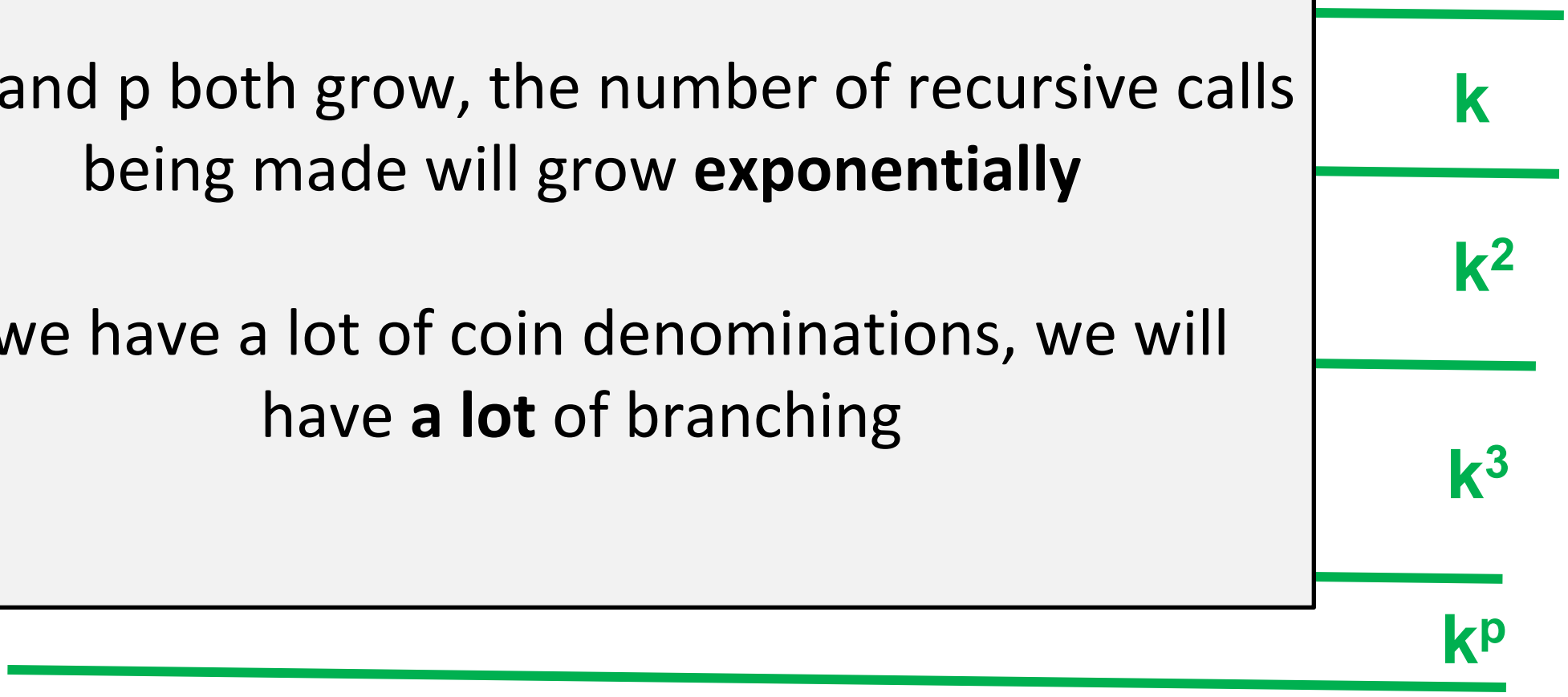
k = # denominations

p = value to make change for

19

As k and p both grow, the number of recursive calls being made will grow **exponentially**

If we have a lot of coin denominations, we will have **a lot** of branching



Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

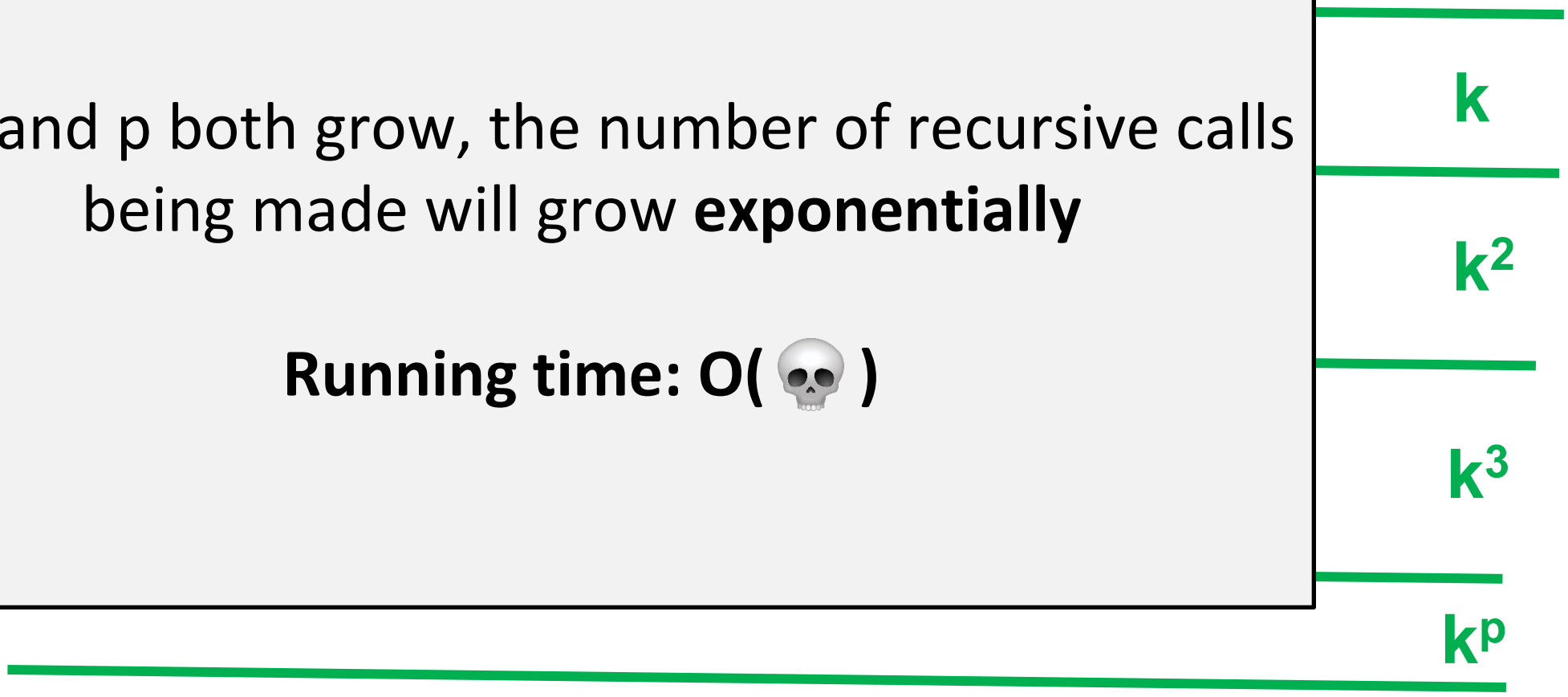
k = # denominations

p = value to make change for

19

As k and p both grow, the number of recursive calls being made will grow **exponentially**

Running time: O(💀)



Change Making Problem

Make $0.19 with $0.01, $0.05, $0.10

k = # denominations

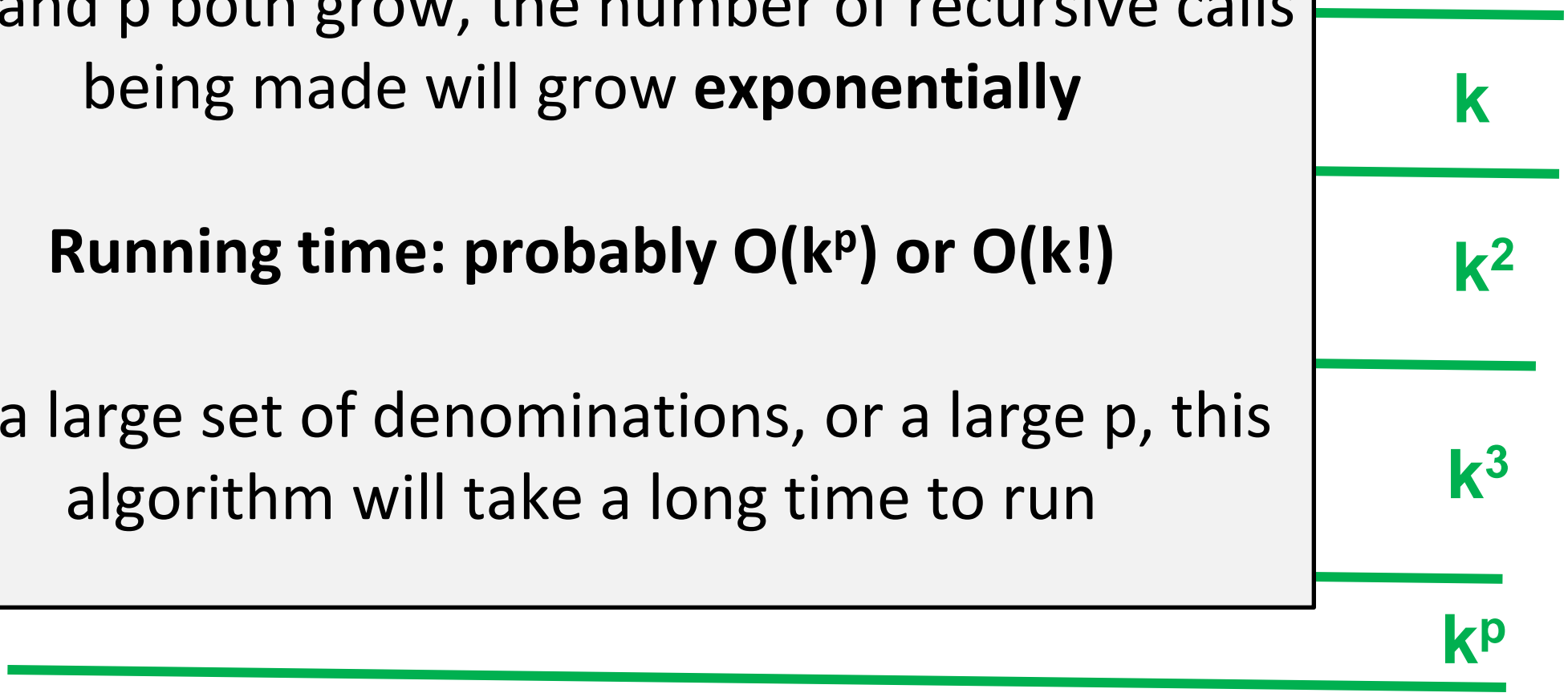
p = value to make change for

19

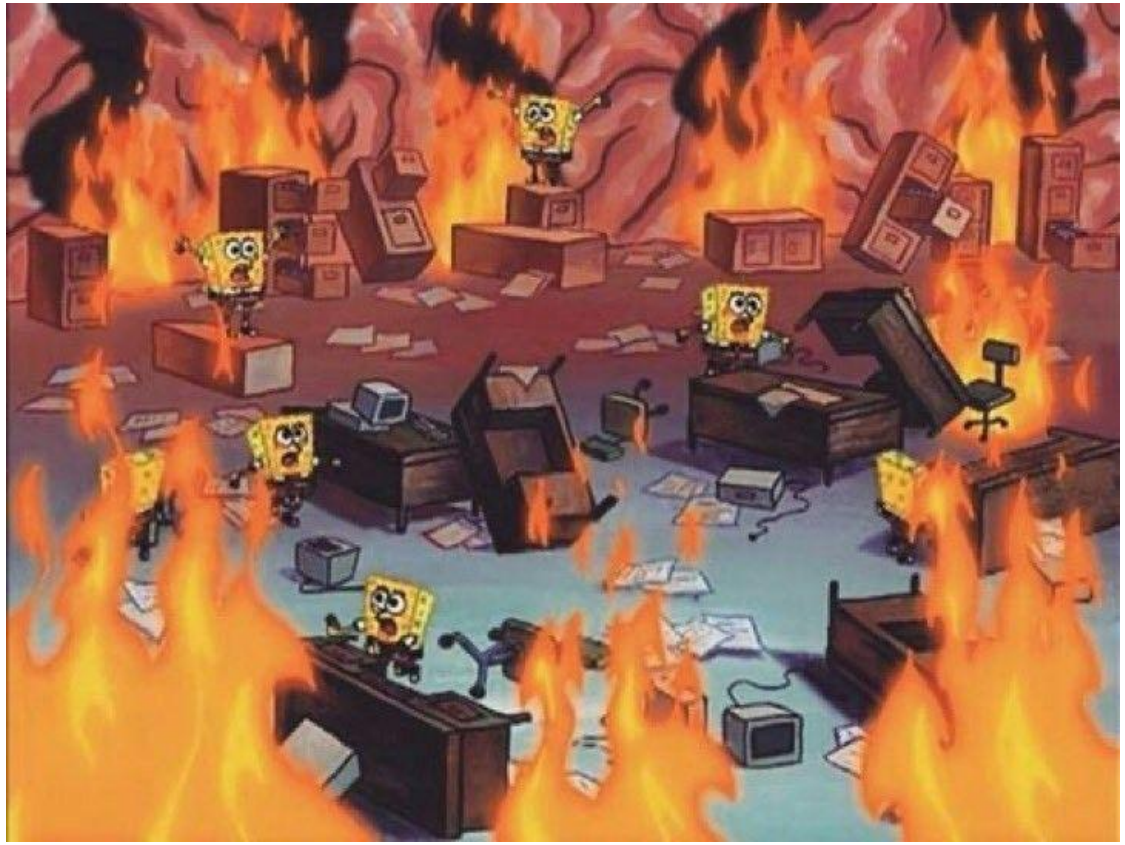
As k and p both grow, the number of recursive calls being made will grow **exponentially**

Running time: probably $O(k^p)$ or $O(k!)$

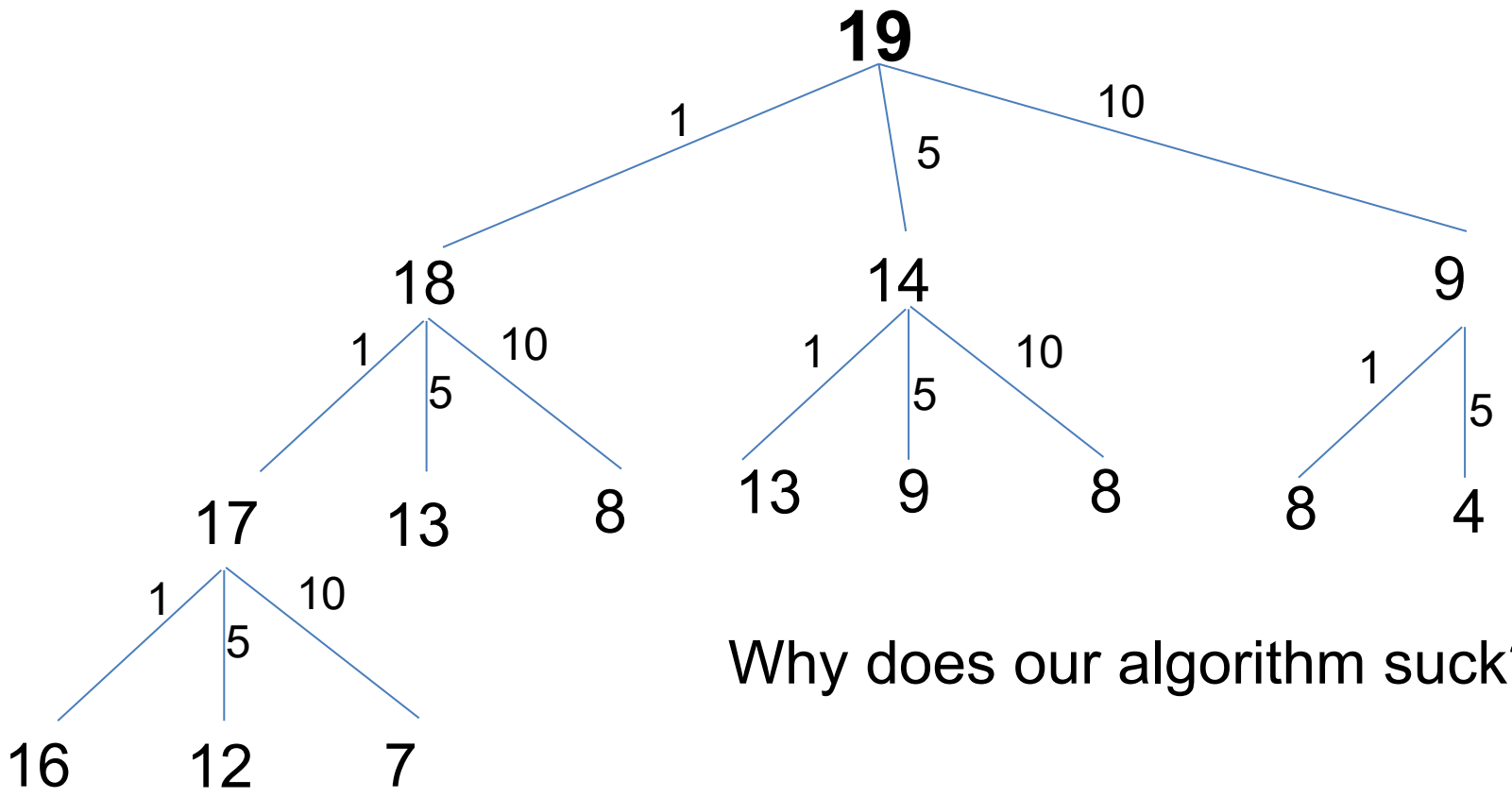
For a large set of denominations, or a large p, this algorithm will take a long time to run



Let’s try 81 cents!

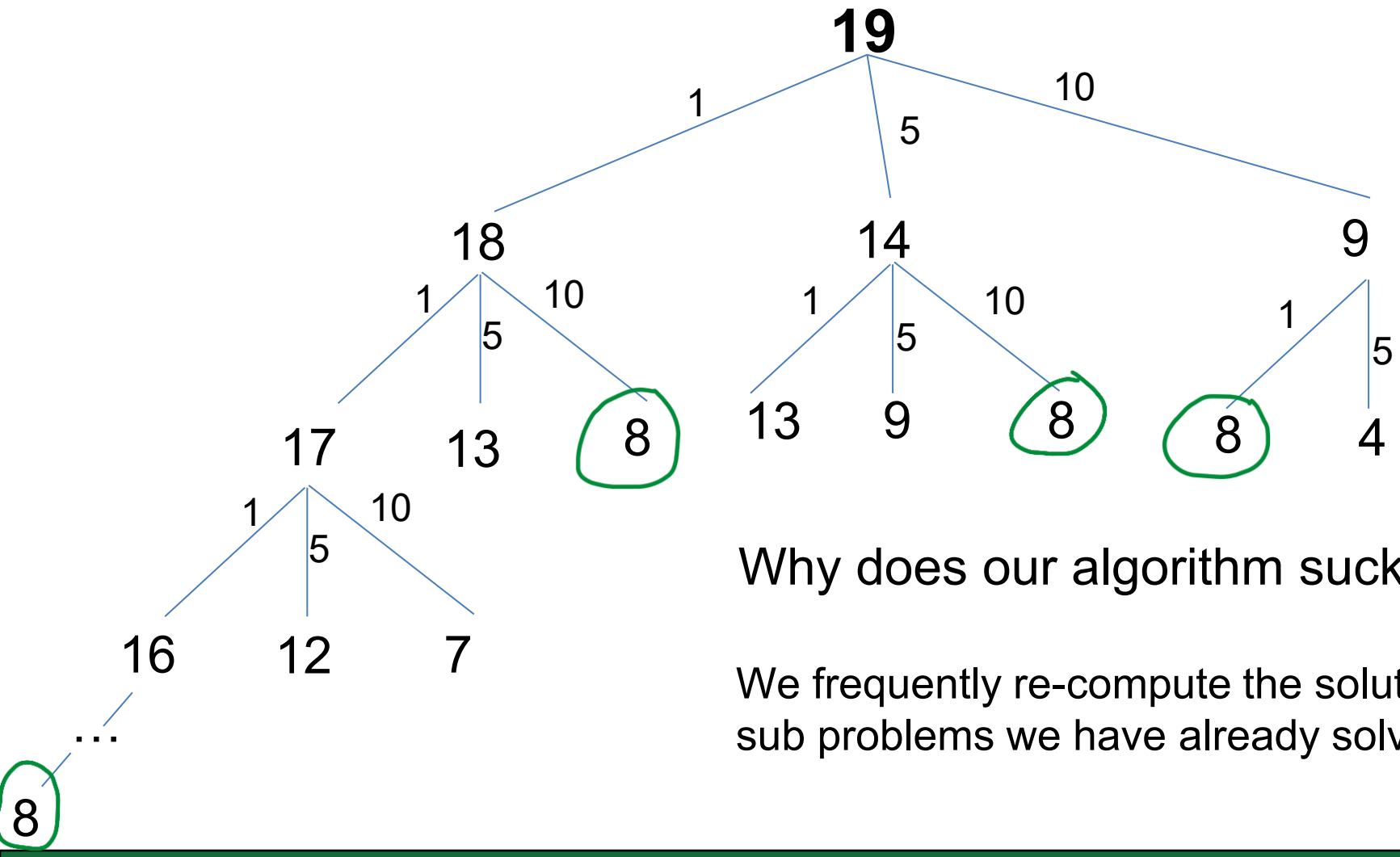


Change Making Problem



Why does our algorithm suck?

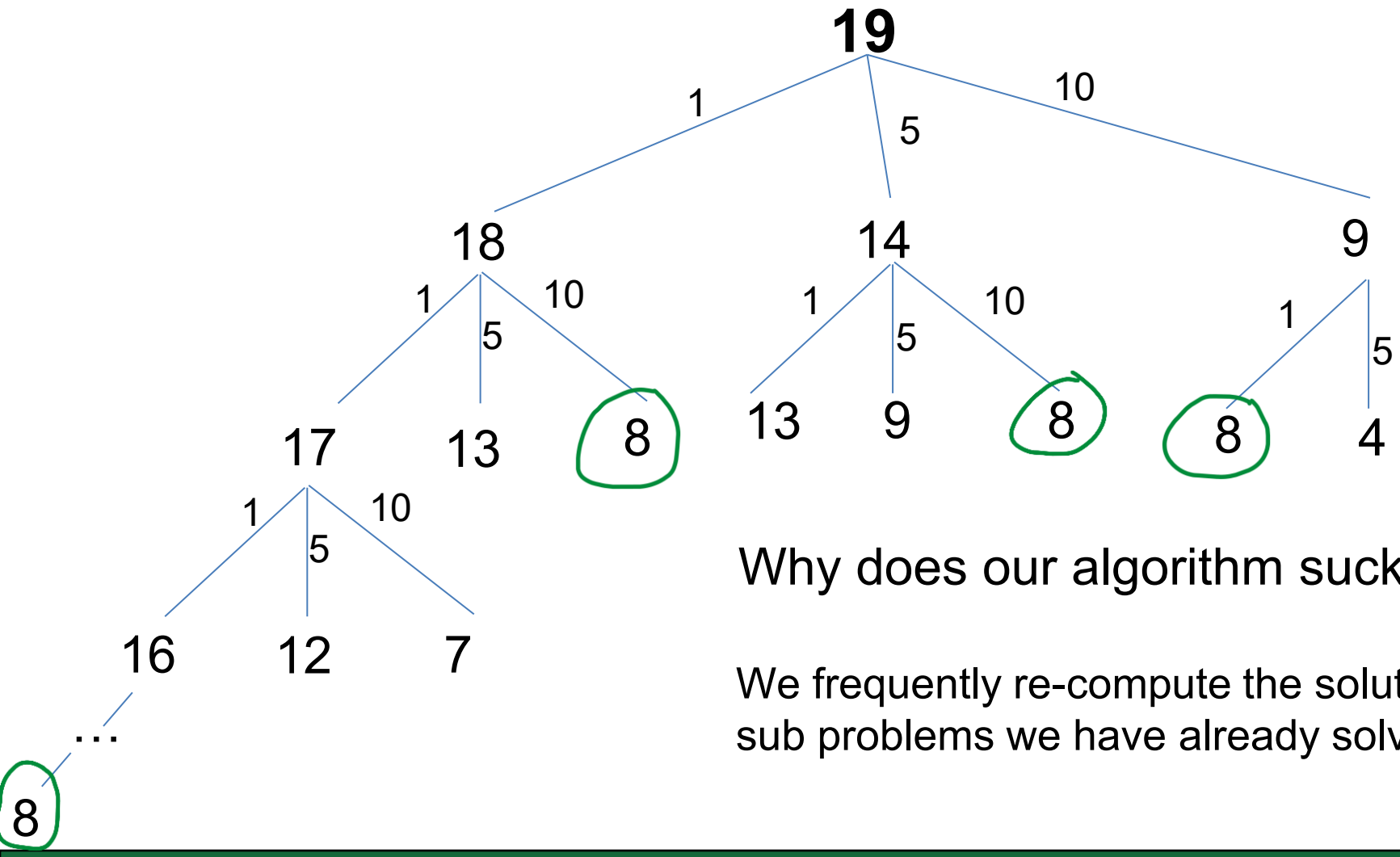
Change Making Problem



Why does our algorithm suck?

We frequently re-compute the solution to the sub problems we have already solved

Change Making Problem



We can fix this by utilizing some “smart recursion” AKA **Dynamic Programming**

Why does our algorithm suck?

We frequently re-compute the solution to the sub problems we have already solved

Dynamic Programming

Dynamic Programming is an algorithm technique used for optimization problems that involves smartly using recursion to solve a problem with many overlapping subproblems

Dynamic Programming

Dynamic Programming is an algorithm technique used for optimization problems that involves smartly using recursion to solve a problem with many overlapping subproblems

To use dynamic programming, we must first identify two characteristics of some problem

(If it has these two characteristics, we can use DP to solve it)

Dynamic Programming

Dynamic Programming is an algorithm technique used for optimization problems that involves smartly using recursion to solve a problem with many overlapping subproblems

To use dynamic programming, we must first identify two characteristics of some problem

(If it has these two characteristics, we can use DP to solve it)

Optimal substructure- an optimal solution can be constructed from optimal solutions of its sub problems

Dynamic Programming

Dynamic Programming is an algorithm technique used for optimization problems that involves smartly using recursion to solve a problem with many overlapping subproblems

To use dynamic programming, we must first identify two characteristics of some problem

(If it has these two characteristics, we can use DP to solve it)

Optimal substructure- an optimal solution can be constructed from optimal solutions of its sub problems

Overlapping Subproblems- we solve the same subproblem several times during the algorithm

Dynamic Programming

Optimal substructure- an optimal solution can be constructed from optimal solutions of its sub problems

The solution to the change making problem consists of solution to smaller change making problems

Overlapping Subproblems- we solve the same subproblem several times during the algorithm

We frequently recompute the same subproblem throughout the algorithm

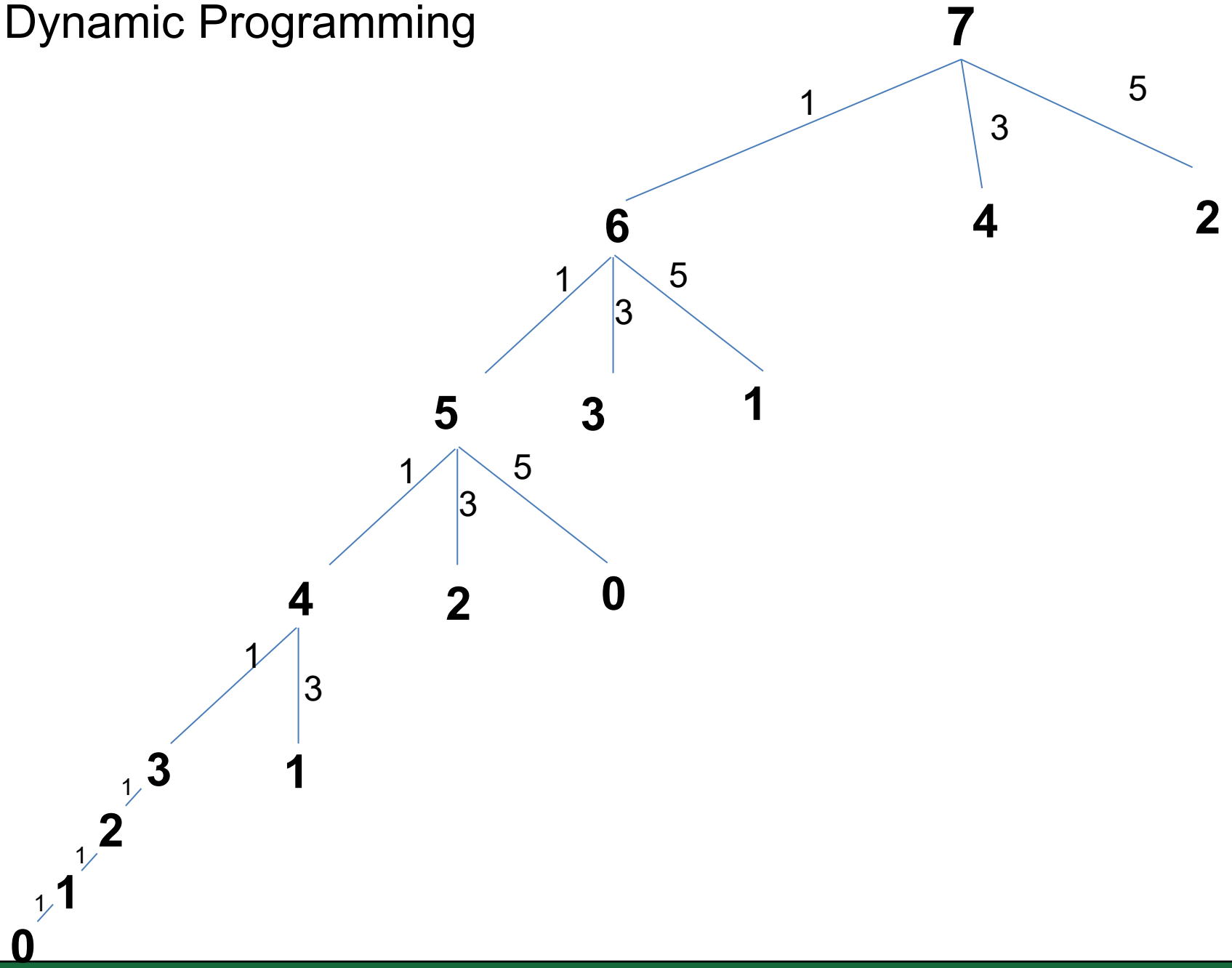
We satisfy these two conditions, which means we can leverage **Dynamic Programming**

Big idea of dynamic programming:

Use **memoization** to store solutions of sub problems we have already solved, and don't re compute them

(Yes, it's "memoization" and not "memorization")

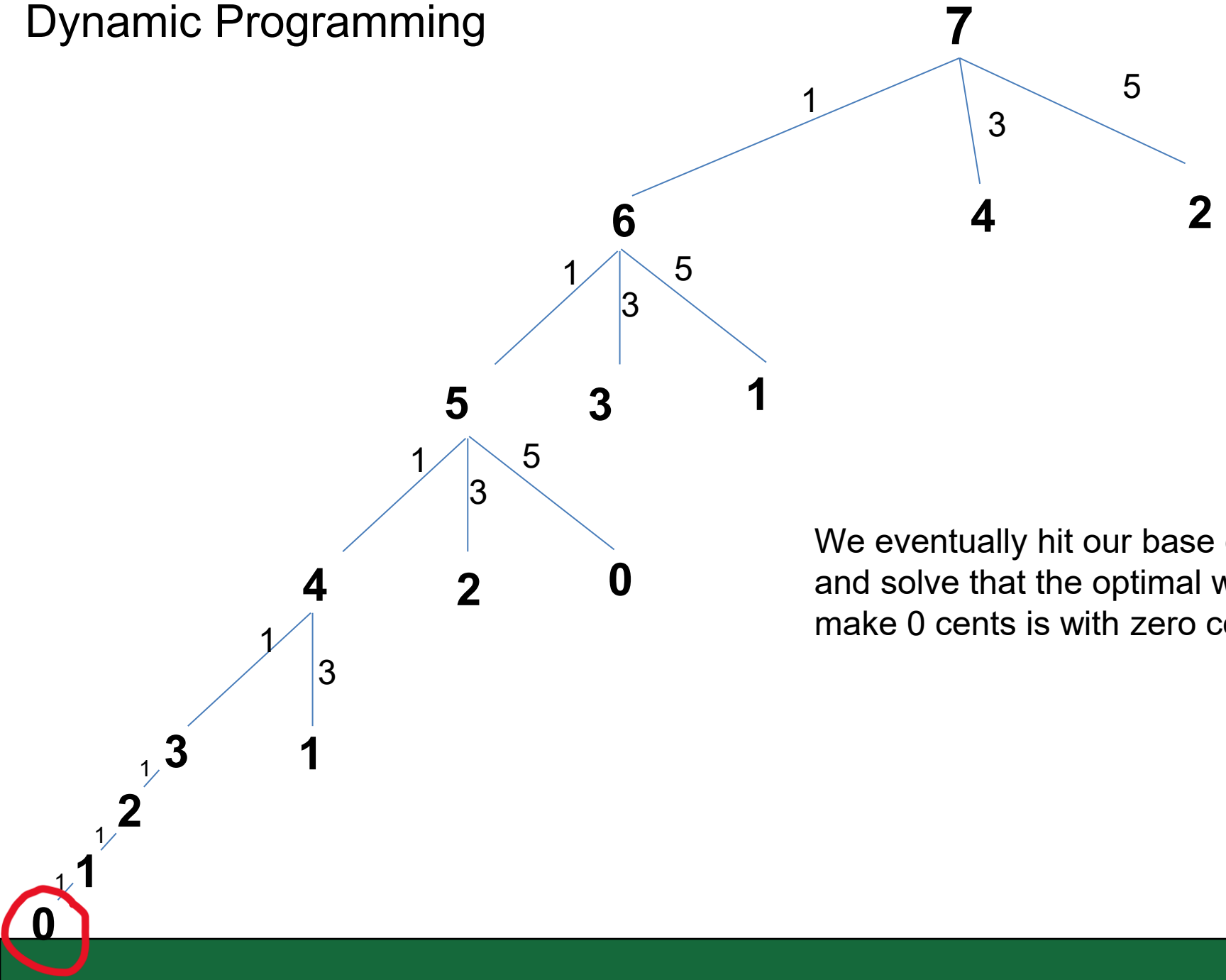
Dynamic Programming



Memoization Table

0	
1	
2	
3	
4	
5	
6	
7	

Dynamic Programming

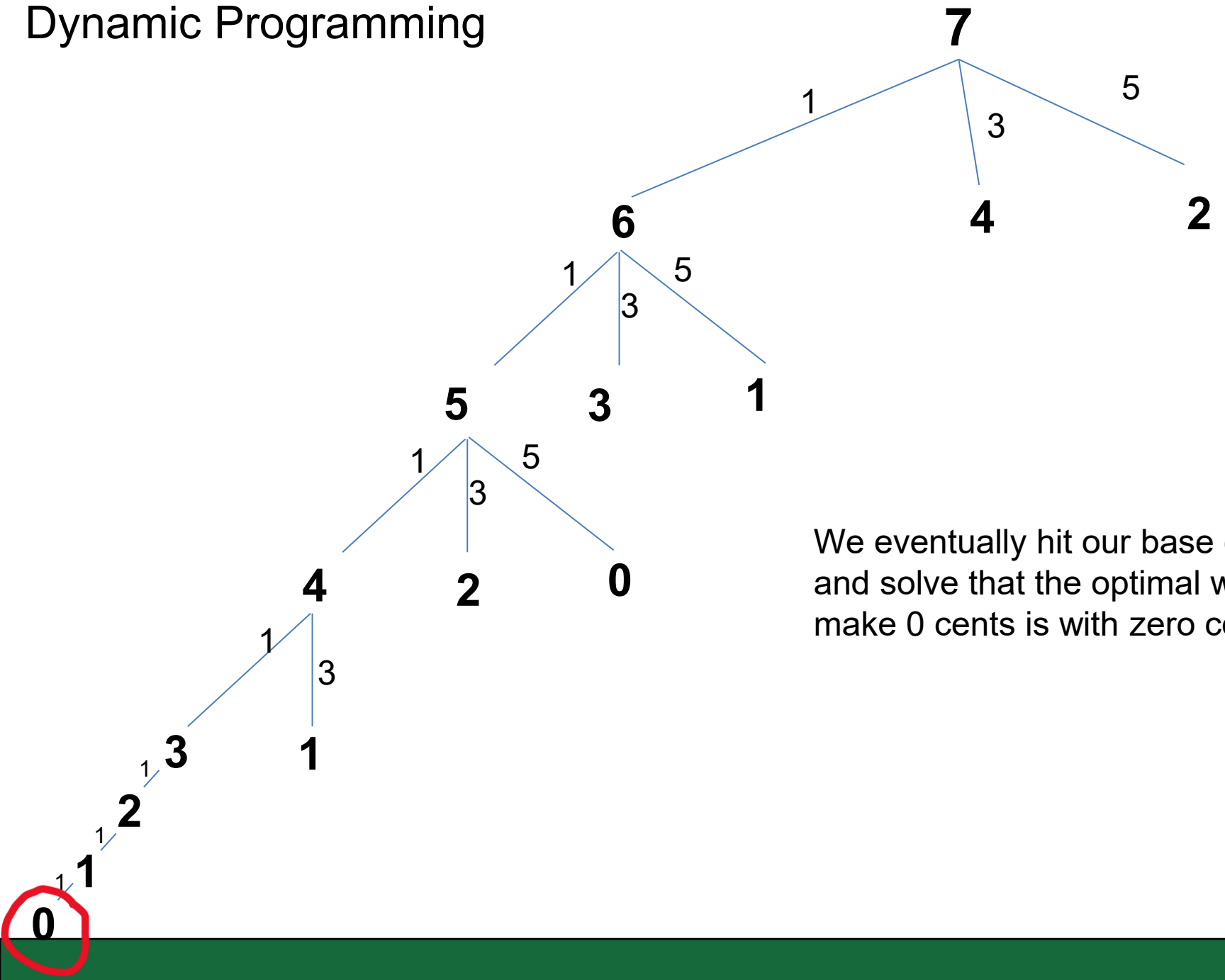


Memoization Table

0	
1	
2	
3	
4	
5	
6	
7	

We eventually hit our base case, and solve that the optimal way to make 0 cents is with zero coins

Dynamic Programming

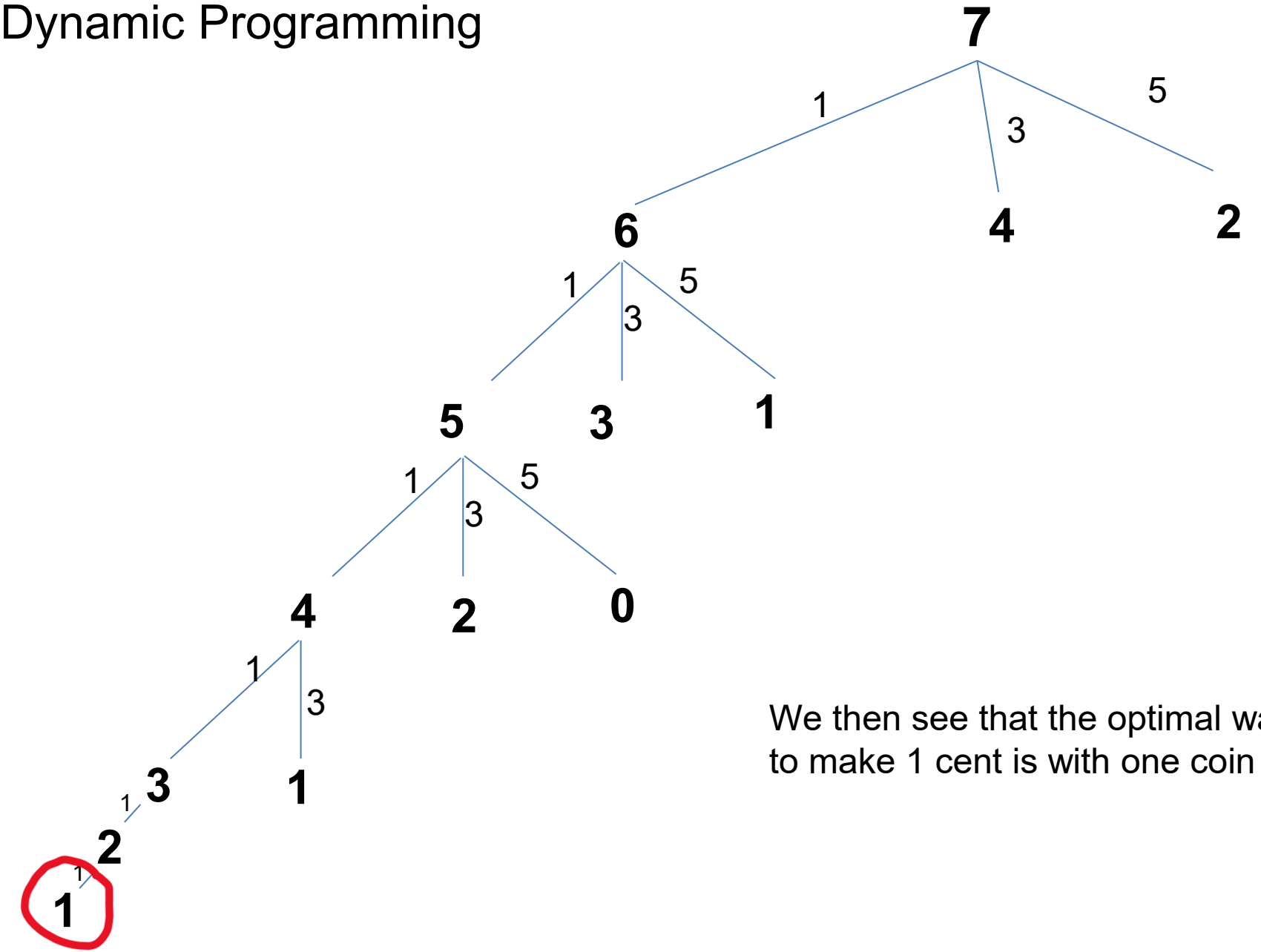


Memoization Table

0	0
1	
2	
3	
4	
5	
6	
7	

We eventually hit our base case, and solve that the optimal way to make 0 cents is with zero coins

Dynamic Programming

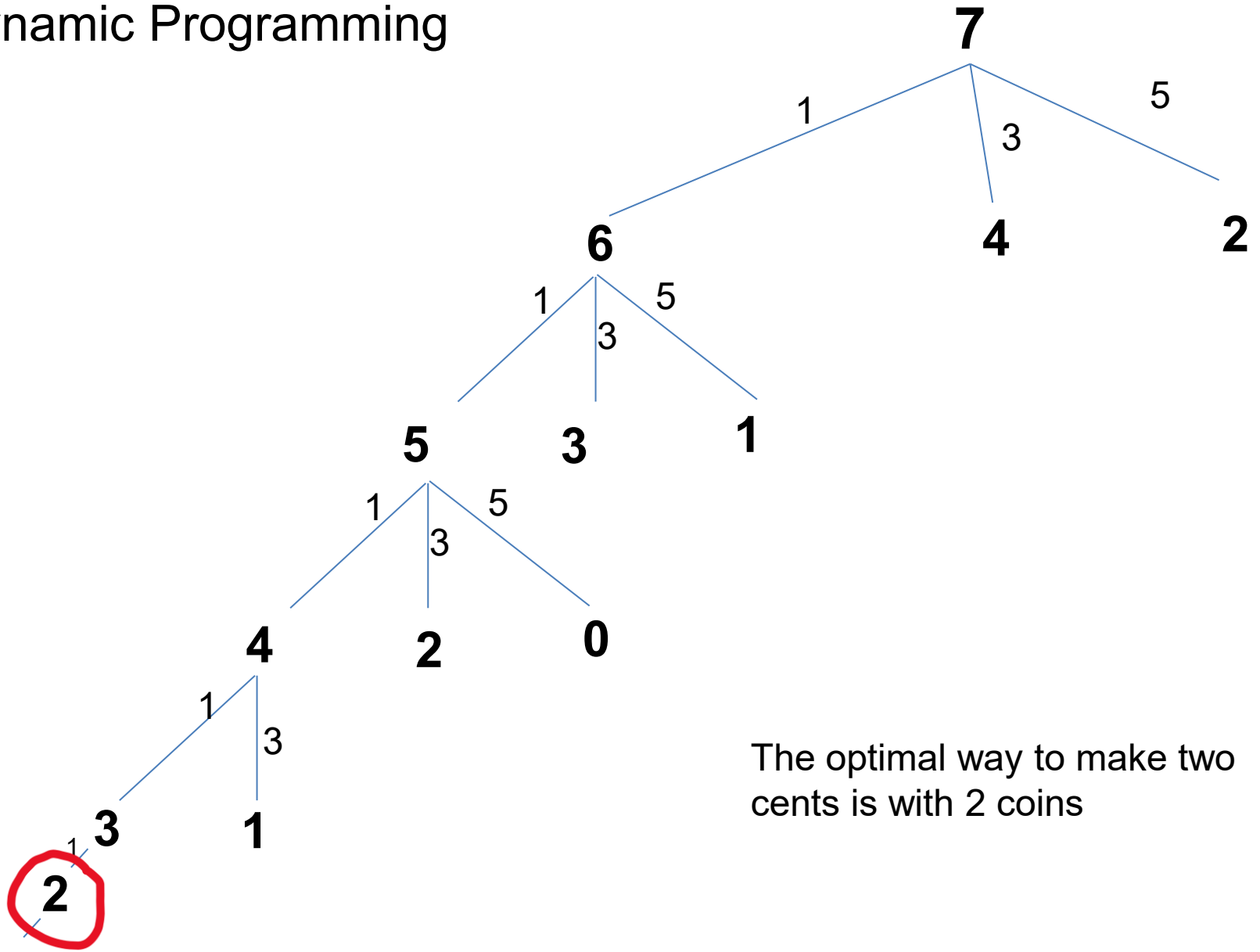


Memoization Table

0	0
1	1
2	
3	
4	
5	
6	
7	

We then see that the optimal way to make 1 cent is with one coin

Dynamic Programming

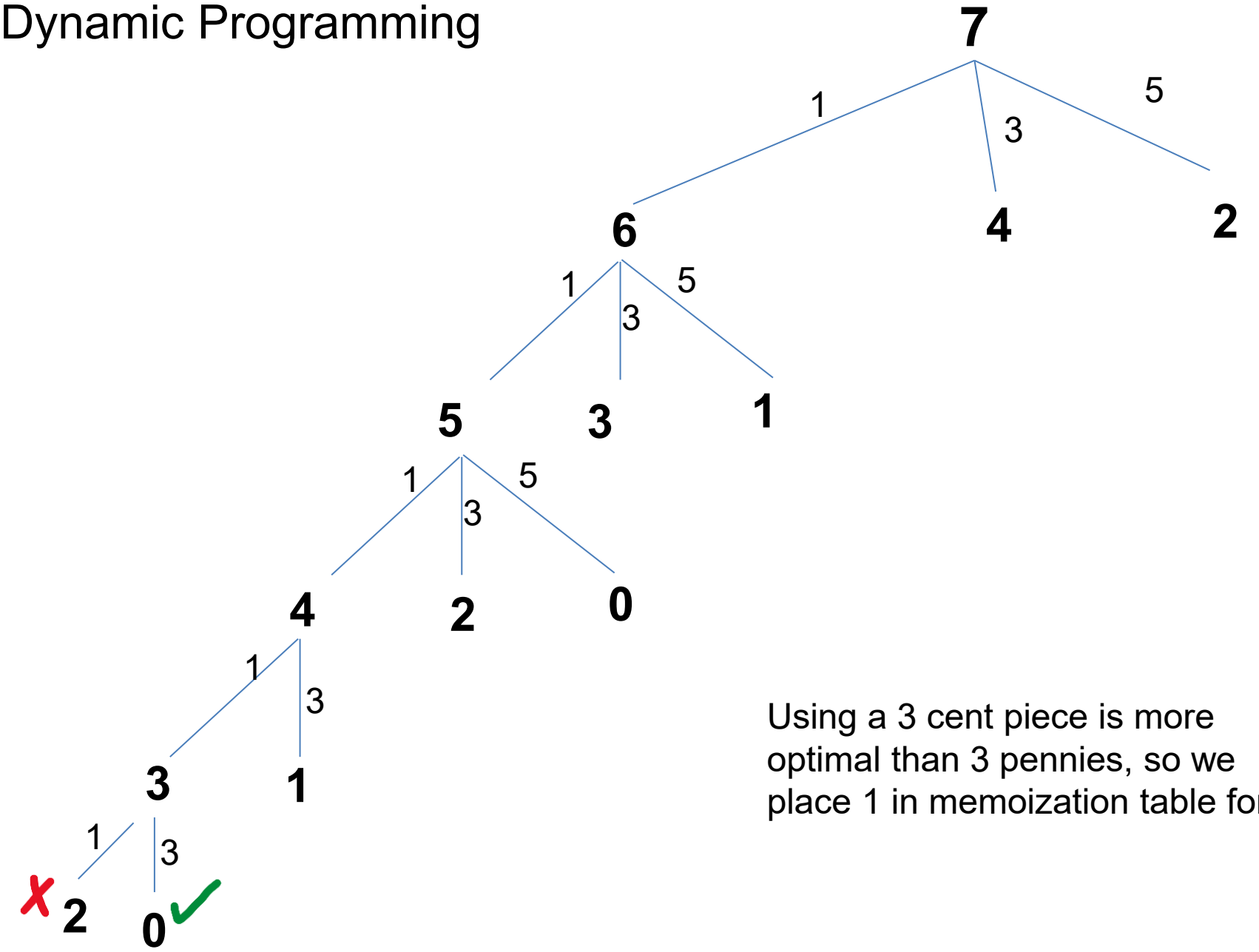


Memoization Table

0	0
1	1
2	2
3	
4	
5	
6	
7	

The optimal way to make two cents is with 2 coins

Dynamic Programming

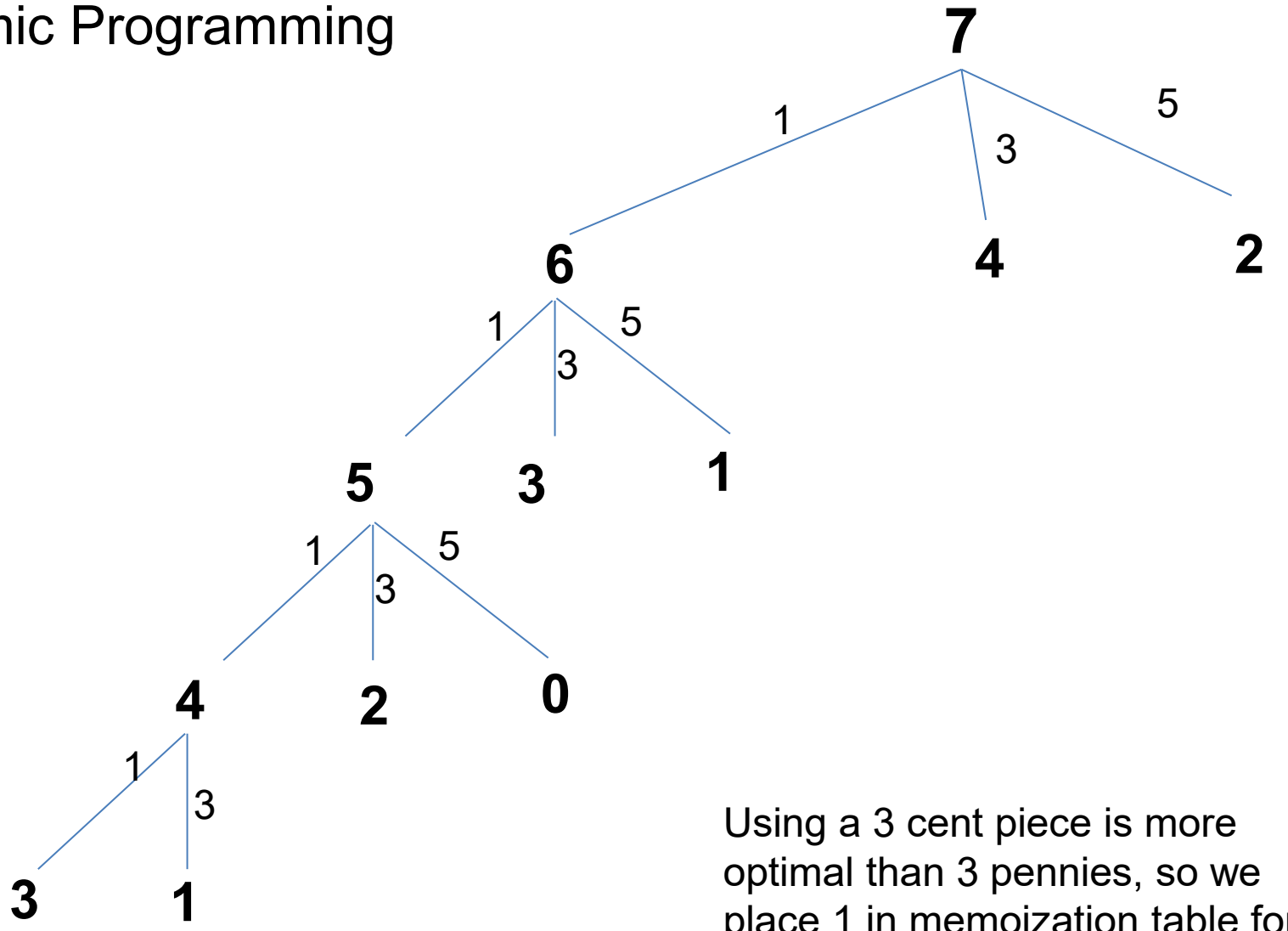


Memoization Table

0	0
1	1
2	2
3	
4	
5	
6	
7	

Using a 3 cent piece is more optimal than 3 pennies, so we place 1 in memoization table for 3

Dynamic Programming

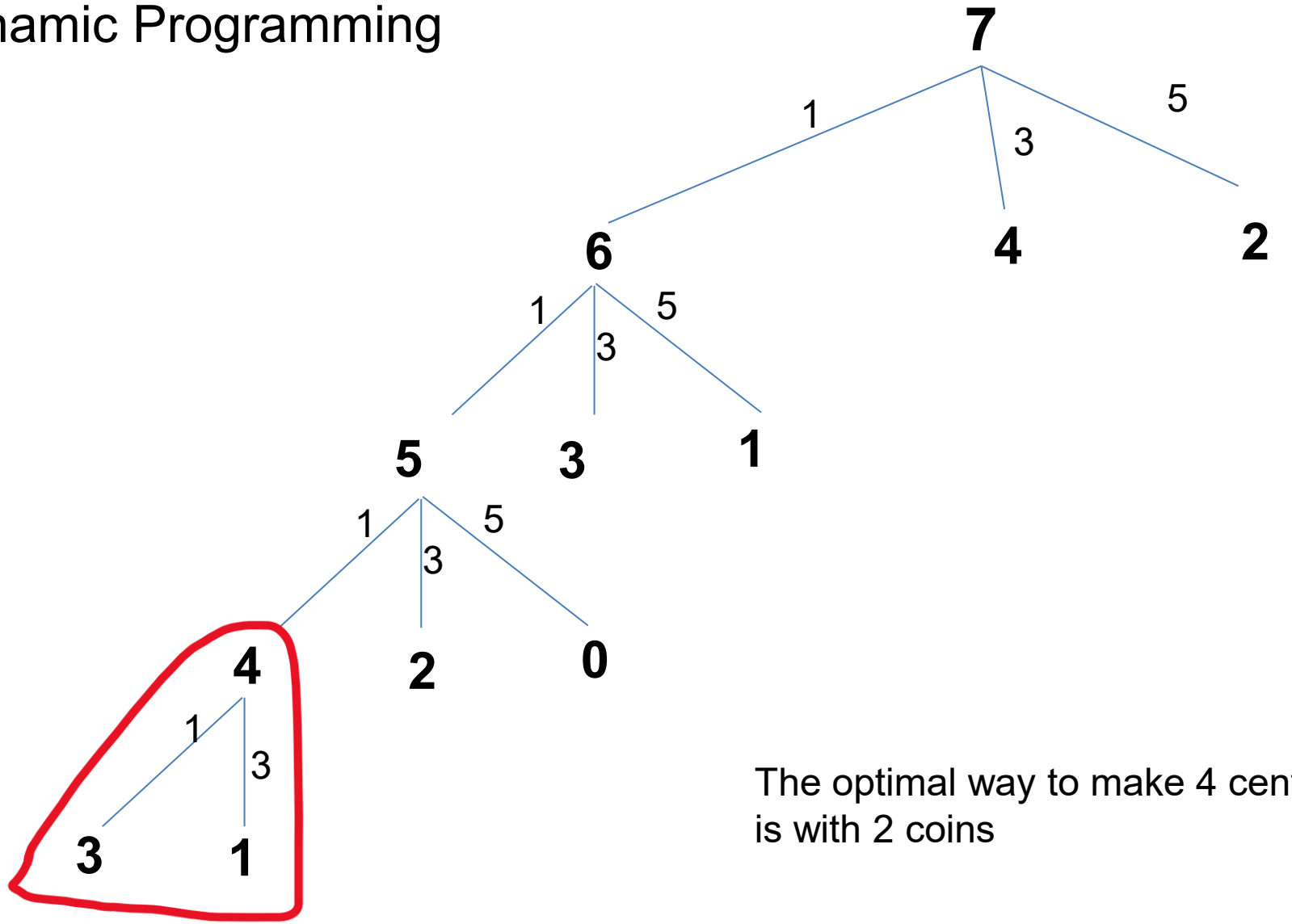


Memoization Table

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

Using a 3 cent piece is more optimal than 3 pennies, so we place 1 in memoization table for 3

Dynamic Programming

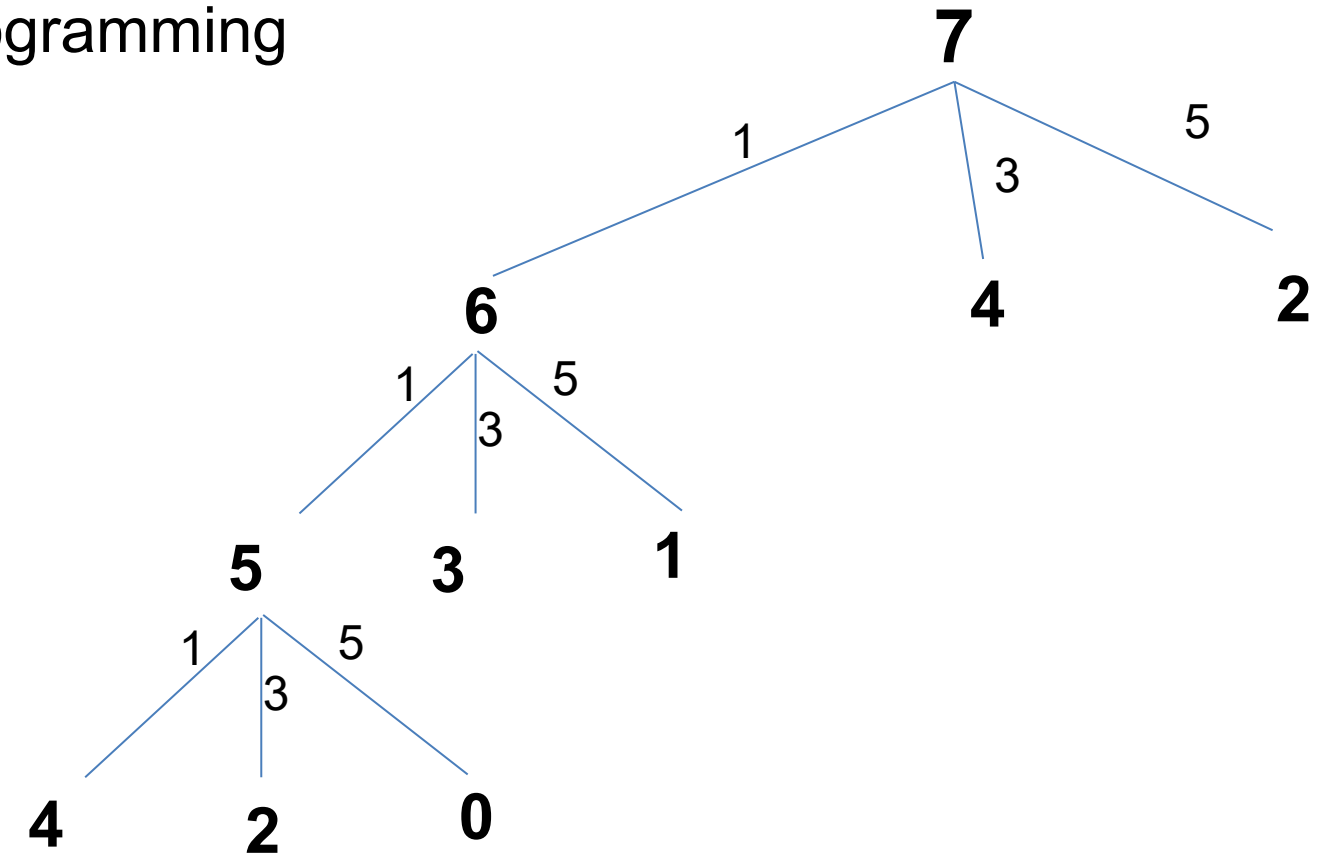


Memoization Table

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

The optimal way to make 4 cents is with 2 coins

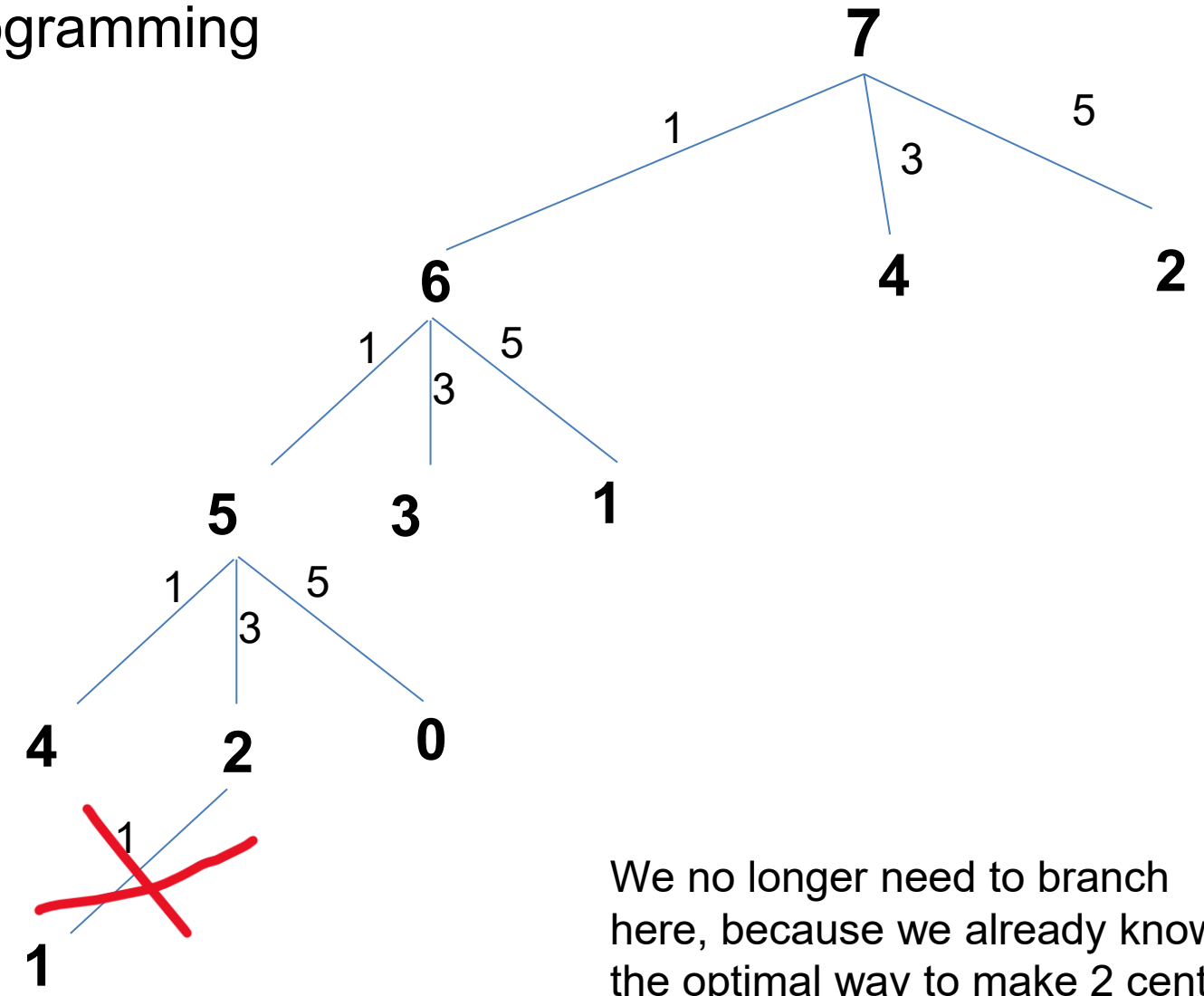
Dynamic Programming



Memoization Table

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

Dynamic Programming

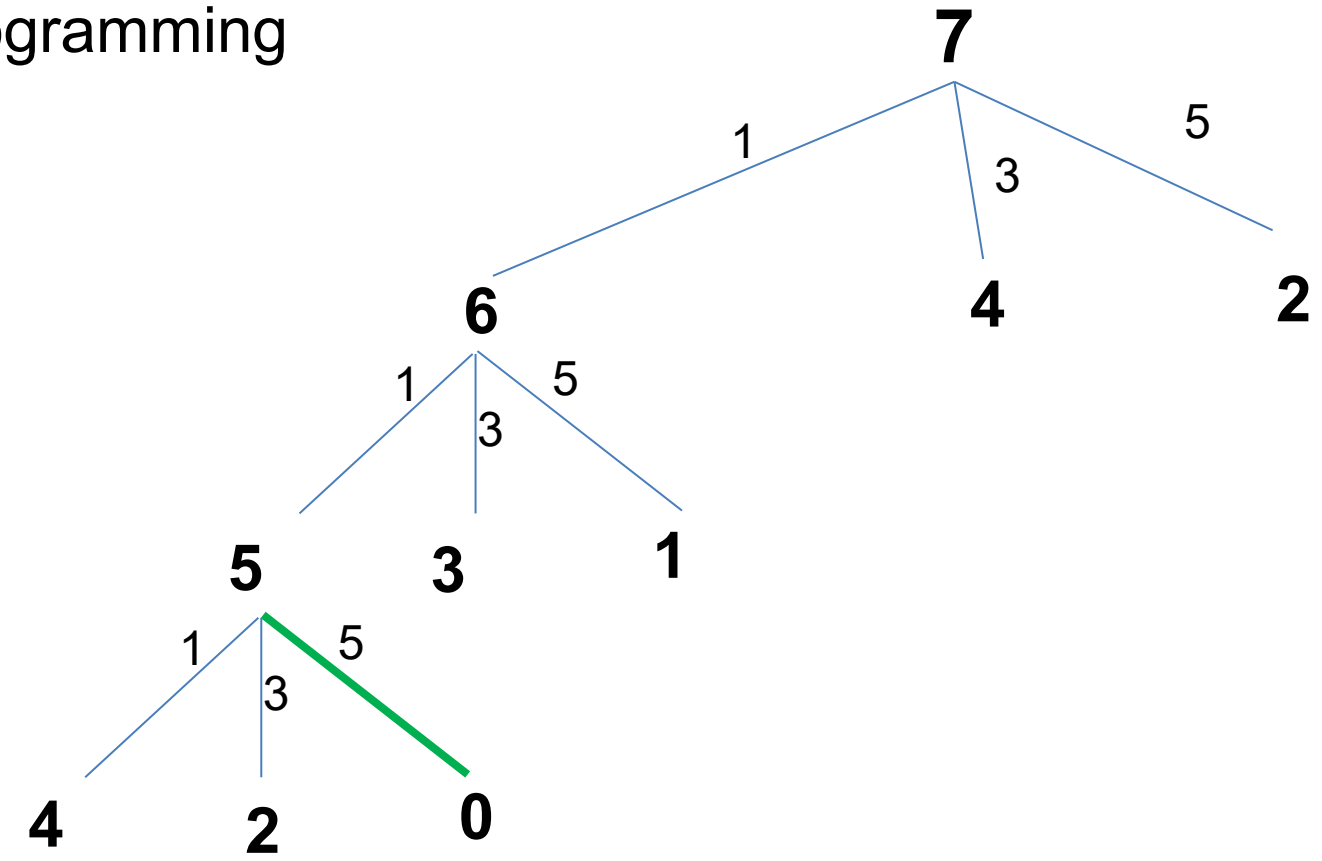


Memoization Table

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

We no longer need to branch here, because we already know the optimal way to make 2 cents!

Dynamic Programming

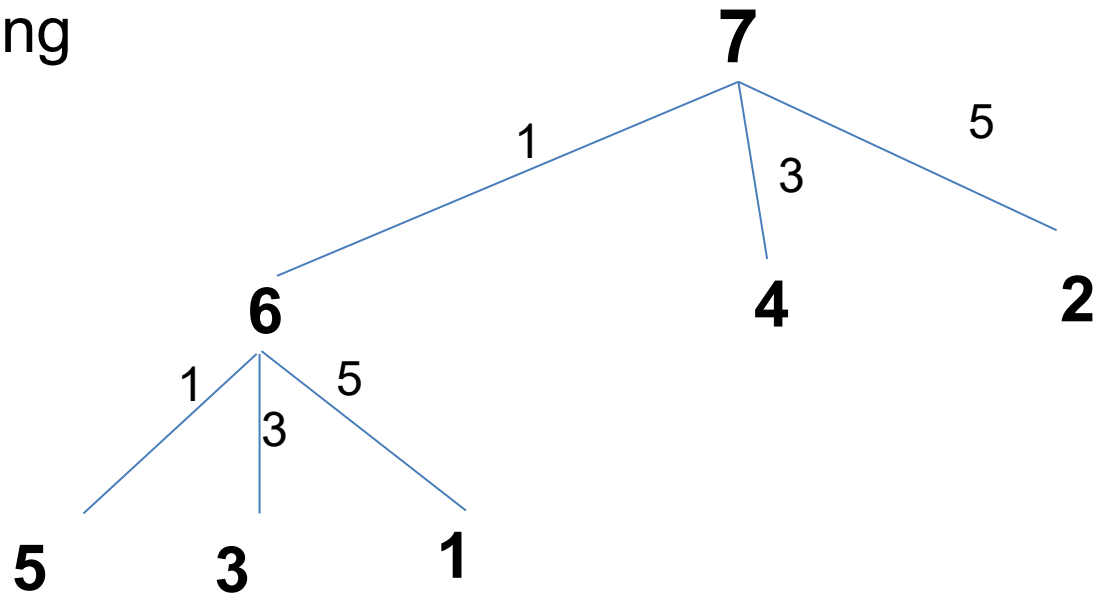


Memoization Table

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

We learn the optimal way to make five cents is with one coin

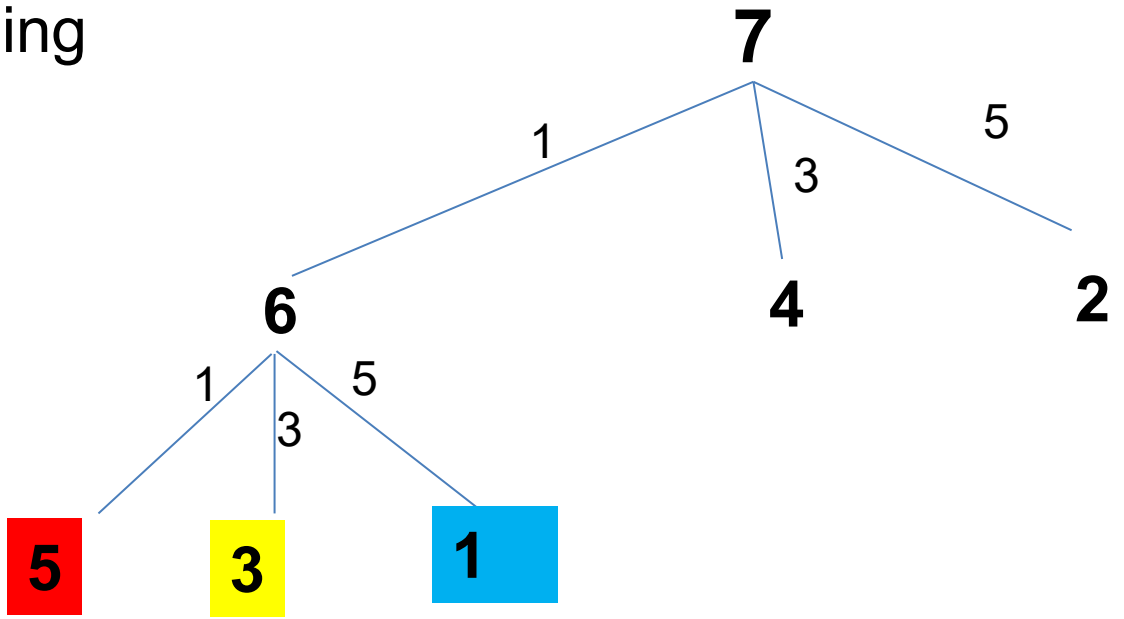
Dynamic Programming



Memoization Table

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

Dynamic Programming

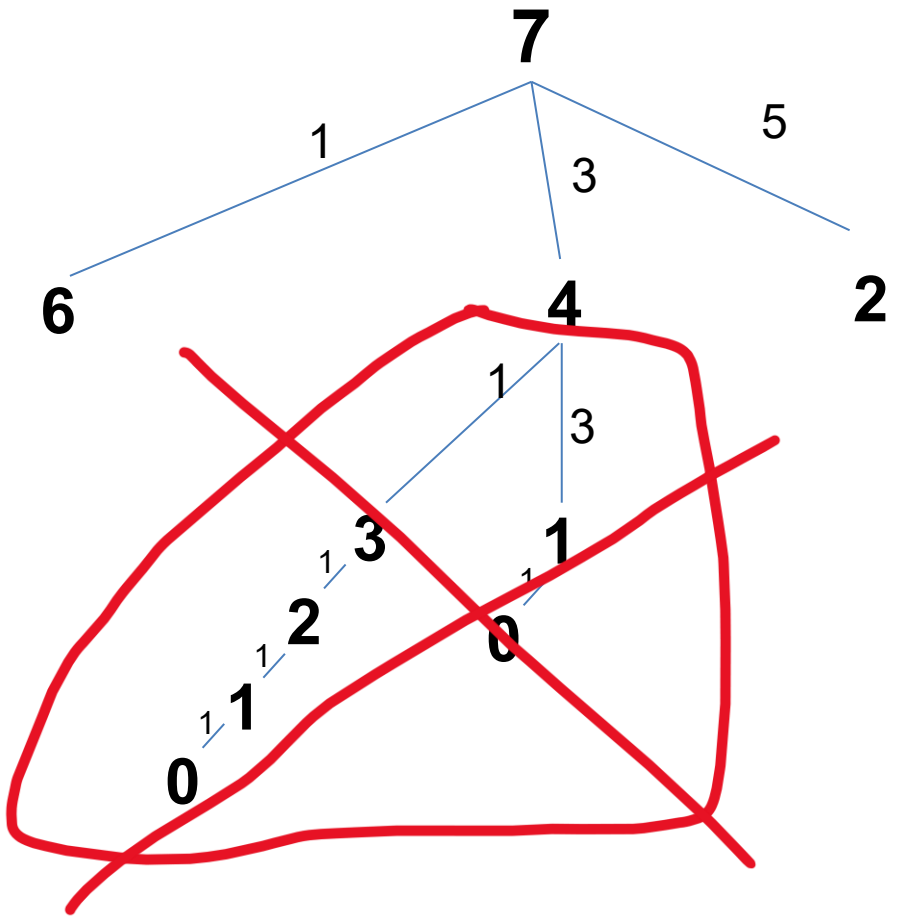


Memoization Table

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

All three of these branches have the same cost, so making 6 cents requires 2 coint

Dynamic Programming

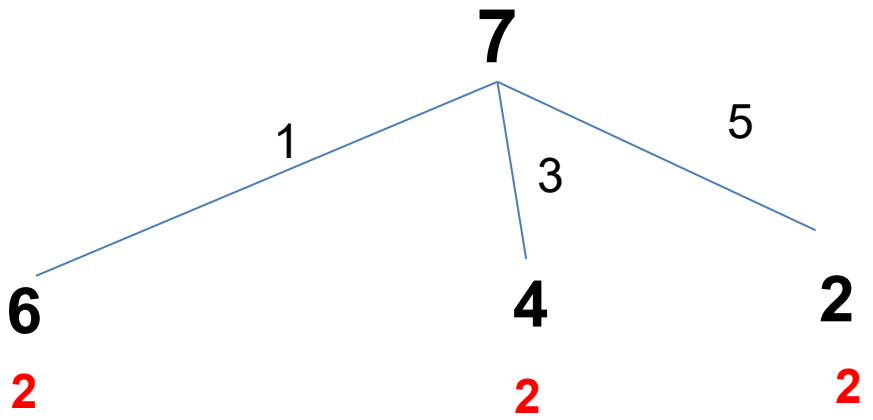


Memoization Table

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

We no longer need to branch here, because we already know the optimal solution for 4, so just check our memoization table!

Dynamic Programming



Memoization Table

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

The optimal way to make 7 cents is with 3 coins:

[1, 1, 5]

[3, 1, 3]

Code

Memoization Table

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

Top – Bottom Dynamic Programming (Memoization)
(Use Recursion)

Memoization Table

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

Dynamic Programming (Bottom to Top)

If we don't want to use recursion, we can use a for loop to fill out this entire array (**tabulation**)

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

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

0	0
1	1
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

0	0
1	1
2	2
3	∞
4	∞
5	∞
6	∞
7	∞

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

	Cache
0	0
1	1
2	2
3	1
4	∞
5	∞
6	∞
7	∞

Dynamic Programming (Bottom to Top)

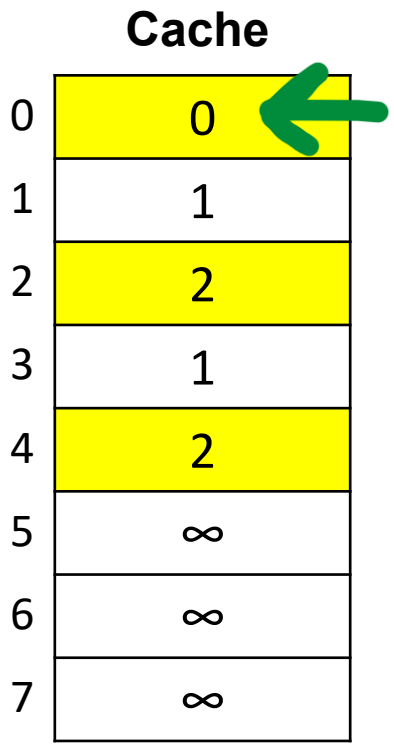
```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

0	0
1	1
2	2
3	1
4	2
5	∞
6	∞
7	∞

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```



Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

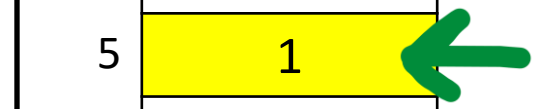
0	0
1	1
2	2
3	1
4	2
5	1
6	∞
7	∞

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

0	0
1	1
2	2
3	1
4	2
5	1
6	∞
7	∞



Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

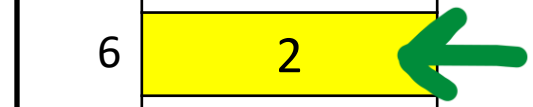
0	0
1	1
2	2
3	1
4	2
5	1
6	2
7	∞

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

0	0
1	1
2	2
3	1
4	2
5	1
6	2
7	∞



Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

Cache

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

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1
```

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

Our table is filled out, we can now query it!

Dynamic Programming (Bottom to Top)

```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c];
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1

return cache[P]
```

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

Our table is filled out, we can now query it!

Dynamic Programming (Bottom to Top)

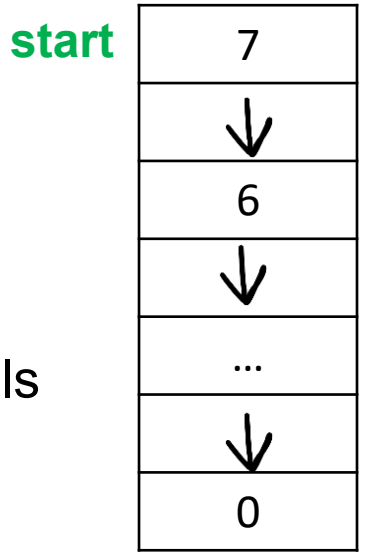
```
cache[0] = 0 //base case
For each cent value, i, in our array (0 – P):
        For each coin in our denomination set c: (1, 3, 5)
                if(c – cache[i] >= 0):
                        x = cache[i – c]
                        if x is smaller than what is currently in the cache:
                                update cache to be x + 1

return ca
```

Bottom-Up Dynamic Programming (Tabulation)

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

Our table is filled out, we can now query it!

Dynamic Programming

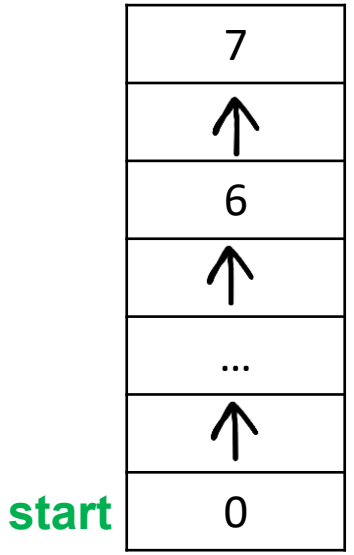


Top to Bottom Dynamic Programming

→ Use recursion, and fill out a **memoization table** as you are making recursive calls

Bottom-Up Dynamic Programming

→ Use a for loop to fill out a table (**tabulation**), then query the table



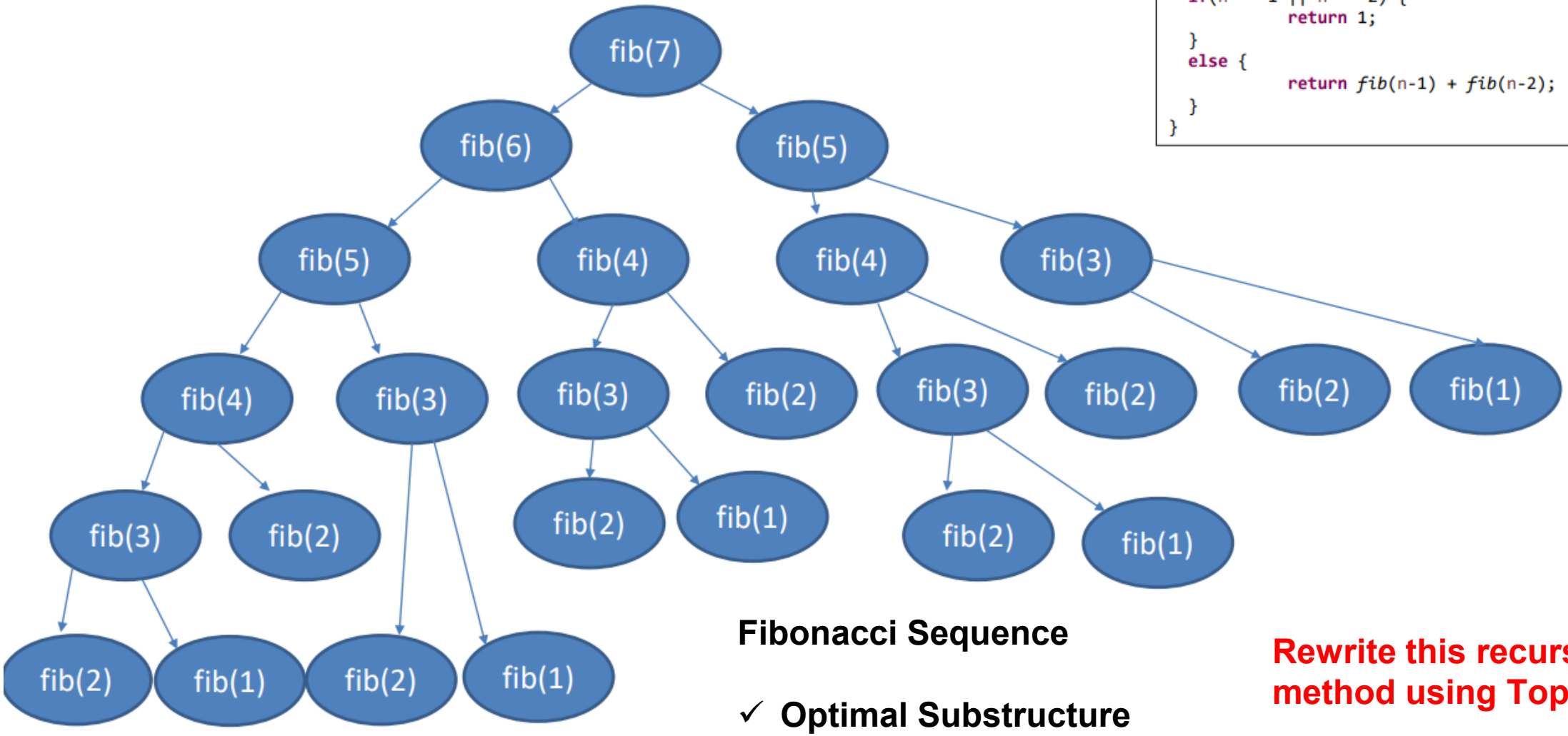
Both have the same running time. But a computer can handle a for loop better than recursion

(I think recursion is easier to understand)

DP improves running time from exponential to **O(len(D) * p)**

Number of recursive calls = 24

```
private static int fib(int n) {
  if(n == 1 || n == 2) {
            return 1;
  }
  else {
            return fib(n-1) + fib(n-2);
  }
}
```



Fibonacci Sequence

- ✓ **Optimal Substructure**
- ✓ **Overlapping Subproblems**

Rewrite this recursive method using Top-Down DP

Edit Distance Problem

Given two strings, how many edits are needed to turn one string into the other?

String 1: **SNOWY**
String 2: **SUNNY**

Edit Distance Problem

Given two strings, how many edits are needed to turn one string into the other?

String 1: **SNOWY**
String 2: **SUNNY**

Another way to state this problem:
What is the minimum cost of aligning these two strings?

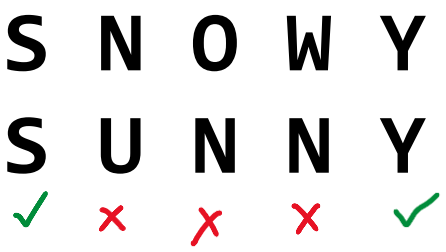
```
S N O W Y
S U N N Y
```

Edit Distance Problem

Given two strings, how many edits are needed to turn one string into the other?

String 1: **SNOWY**
String 2: **SUNNY**

Another way to state this problem:
What is the minimum cost of aligning these two strings?



Cost to align these strings: 3

Edit Distance Problem

Given two strings, how many edits are needed to turn one string into the other?

String 1: **SNOWY**
String 2: **SUNNY**

(It's possible these strings could be different lengths)

Another way to state this problem:
What is the minimum cost of aligning these two strings?

We are allowed to insert "spaces" into the strings

```
S N O W Y        S - N O W Y
S U N N Y        S U N N - Y
✓ × ✗ × ✓
```

Cost to align these strings: 3

Edit Distance Problem

Given two strings, how many edits are needed to turn one string into the other?

String 1: **SNOWY**
String 2: **SUNNY**

(It's possible these strings could be different lengths)

Another way to state this problem:
What is the minimum cost of aligning these two strings?

We are allowed to insert "spaces" into the strings

S	N	O	W	Y
S	U	N	N	Y
✓	×	×	×	✓

Cost to align these strings: 3

S	-	N	O	W	Y
S	U	N	N	-	Y
✓	×	✓	×	×	✓

Cost to align these strings: 3

Edit Distance Problem

Given two strings, how many edits are needed to turn one string into the other?

String 1: **SNOWY**
String 2: **SUNNY**

(It's possible these strings could be different lengths)

Another way to state this problem:
What is the minimum cost of aligning these two strings?

We are allowed to insert "spaces" into the strings

```
S N O W Y
S U N N Y
✓ ✗ ✗ ✗ ✓
```

Cost to align these strings: 3

```
S - N O W Y
S U N N - Y
✓ ✗ ✓ ✗ ✗ ✓
```

Cost to align these strings: 3

```
- S N O W - Y
S U N - - N Y
✗ ✗ ✓ ✗ ✗ ✗ ✓
```

Cost to align these strings: 5

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j) =$ optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j) =$ optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Suppose this is the optimal way to align X and Y

```
S - N O W Y
S U N N - Y
✓ ✗ ✓ ✗ ✗ ✓
```

Cost to align these strings: 3

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j)$ = optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Suppose this is the optimal way to align X and Y

```
S - N O W Y
S U N N - Y
```

Cost to align these strings: 3

```
- N O W Y
U N N - Y
```

then there is not a more optimal way to align NOWY and UNNY in less than 3 edits (otherwise we would have a more optimal way to align SNOWY and SUNNY

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j) =$ optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j)$ = optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:

x_i

\-

Cost: 1

We align a character i from X with a space from Y

S U N N Y

S N O W Y -

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j)$ = optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:

x_i -

\- y_j

Cost: 1 1

We align a character j from Y with a space from X

S U N N Y -

S N O W Y

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j) =$ optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:

	x_i	-	x_i
	-	y_j	y_j
Cost:	1	1	{0,1}

We align a character i from X with character j from Y

S U N N Y
S N O W Y

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j)$ = optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:

	x_i	-	x_i
	-	y_j	y_j
Cost:	1	1	{0,1}

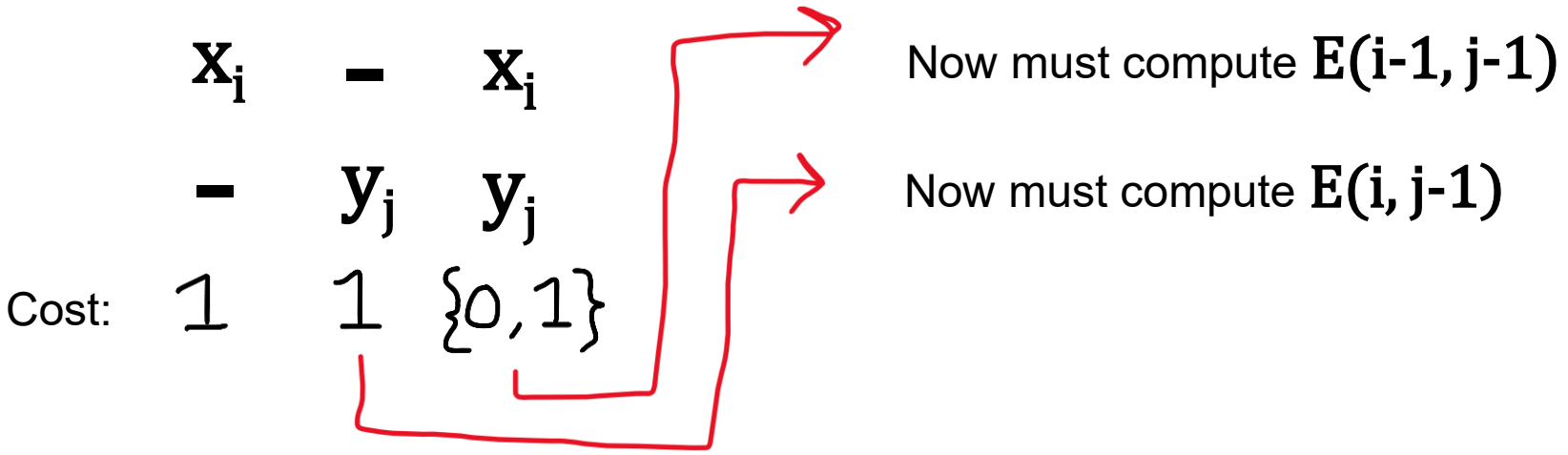
Now must compute $E(i\text{-}1, j\text{-}1)$

Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j)$ = optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:

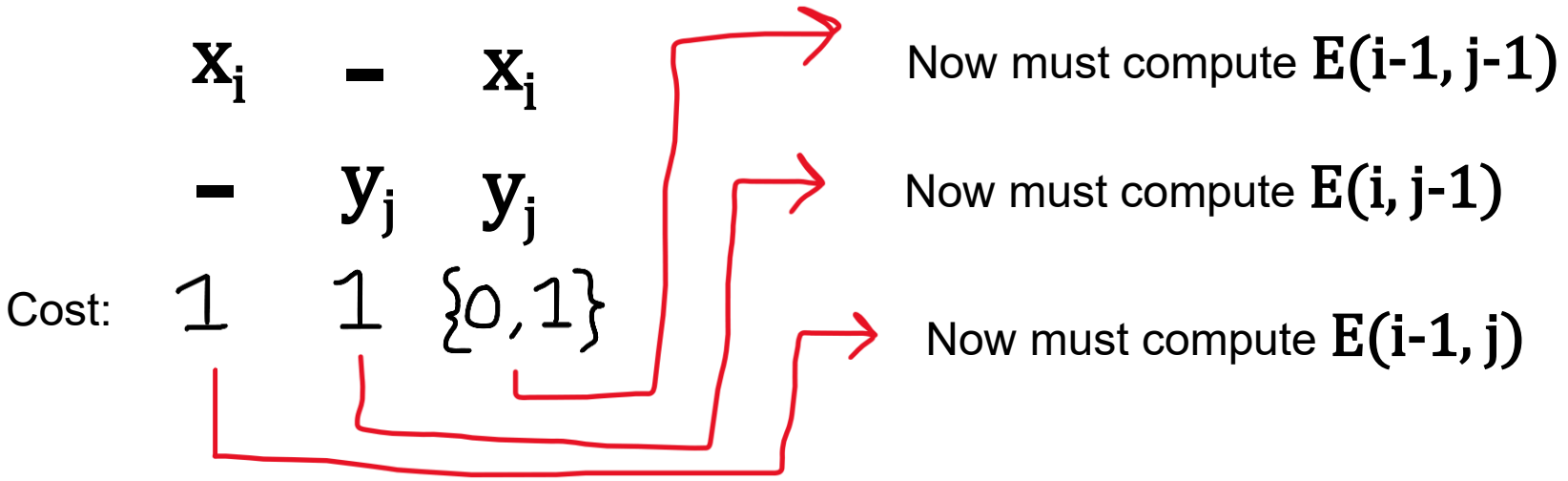


Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j)$ = optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:

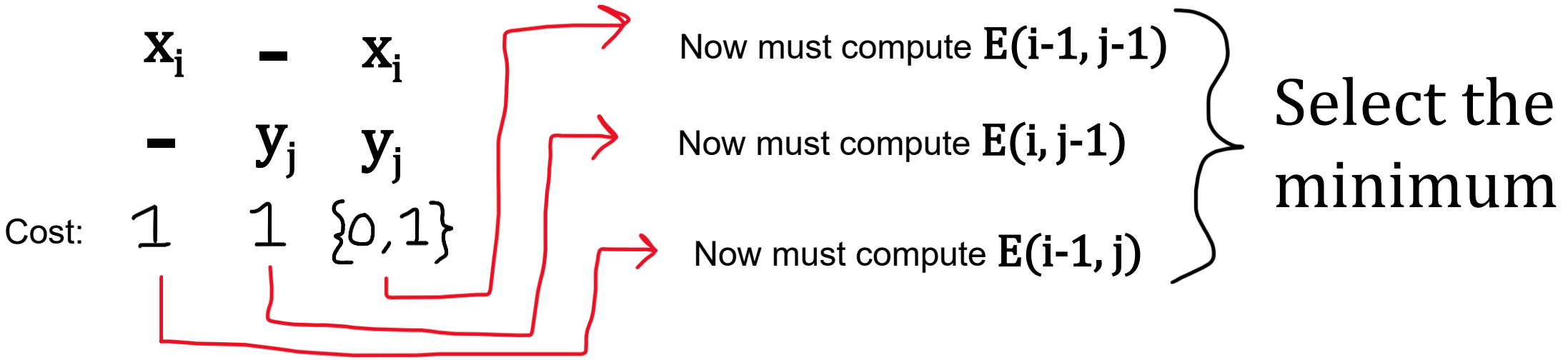


Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j) =$ optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

Optimal alignments end in one of three ways:



Edit Distance Problem

We want to align two strings $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$.

$E(i, j) =$ optimal cost of aligning $[x_1, \ldots, x_i]$ and $[y_1, \ldots, y_j]$.

$$E(i,j) = \min \begin{cases} E(i-1,j) + 1 & x_i \;\; - \\ E(i,j-1) + 1 & - \;\; y_j \\ E(i-1,j-1) + \text{diff}(i,j) & x_i \;\; y_j \end{cases}$$

$$\text{where diff}(i,j) = \begin{cases} 0, & x_i = y_j \\ 1, & x_i \neq y_j \end{cases}$$

S	-	N	O	W	Y
✓	✗	✓	✗	✗	✓
S	U	N	N	-	Y

Cost to align these strings: 3

-	N	O	W	Y
✗	✓	✗	✗	✓
U	N	N	-	Y

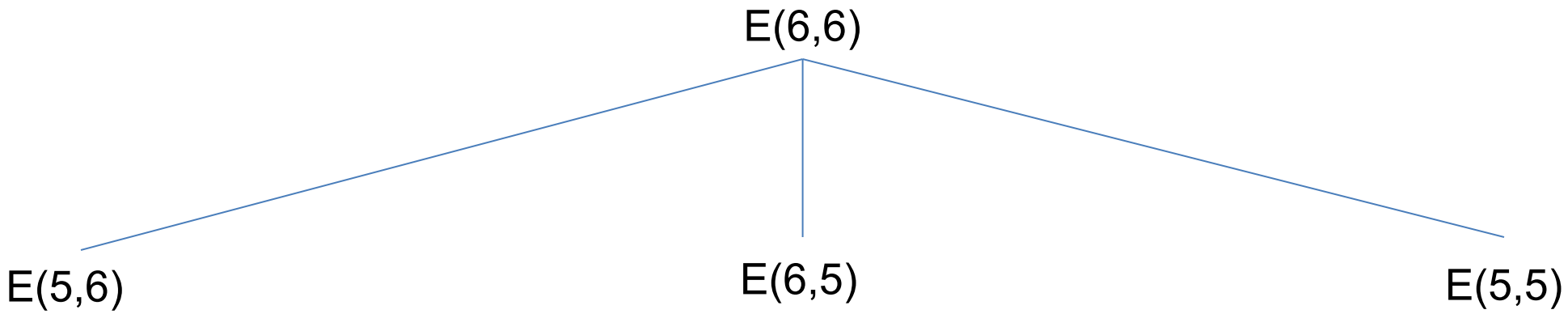
then there is not a more optimal way to align NOWY and UNNY in less than 3 edits (otherwise we would have a more optimal way to align SNOWY and SUNNY

Optimal Substructure

Edit Distance Problem

Finding **E(n,m)** requires finding all other **E**'s

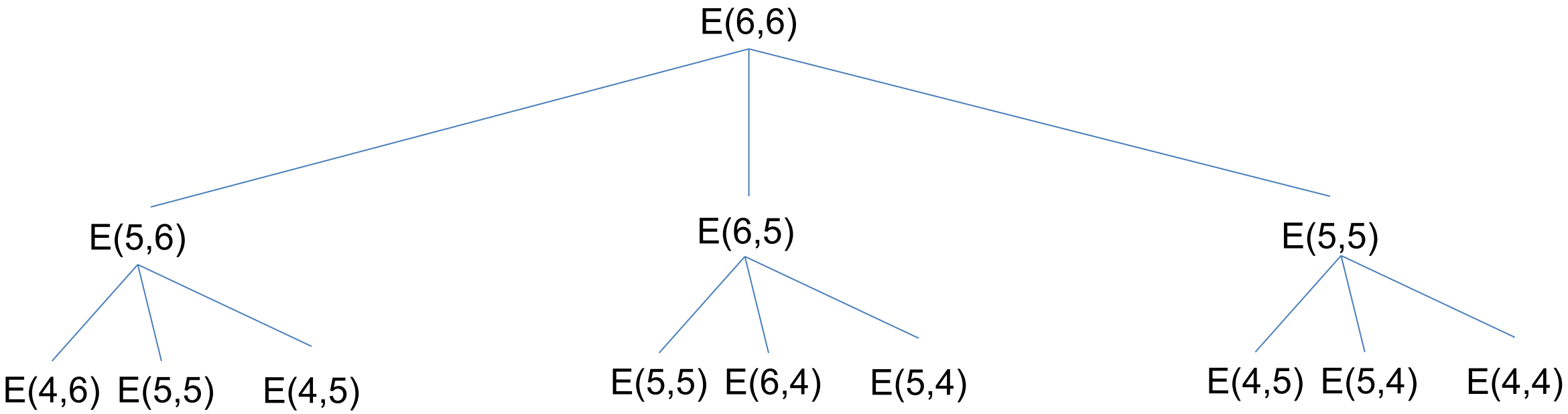
$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$



Edit Distance Problem

Finding **E(n,m)** requires finding all other **E**'s

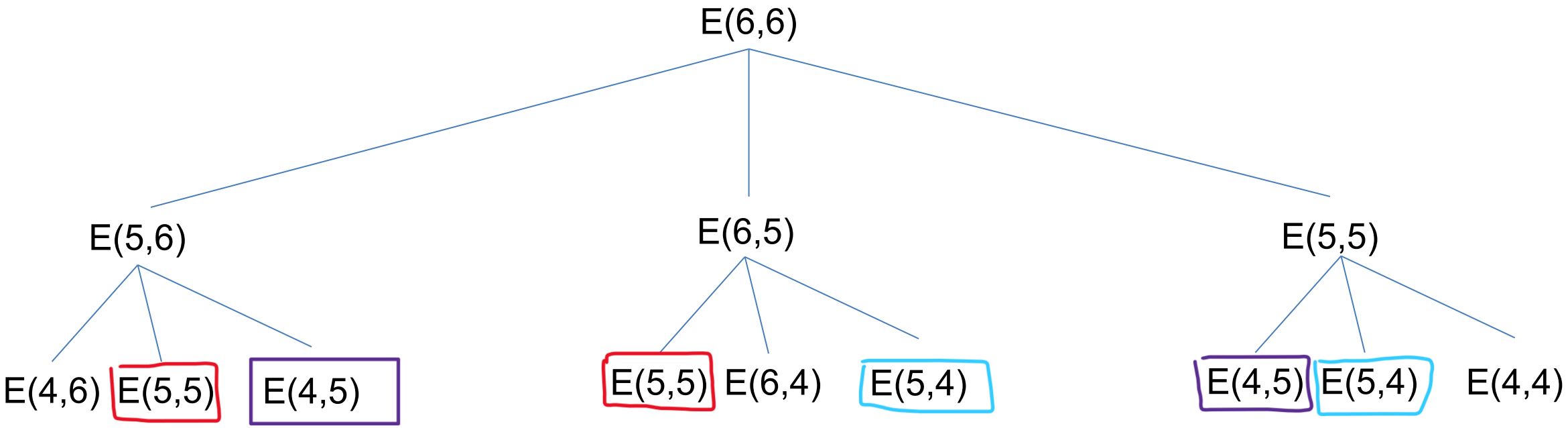
$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$



Edit Distance Problem

Finding **E(n,m)** requires finding all other **E**'s

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

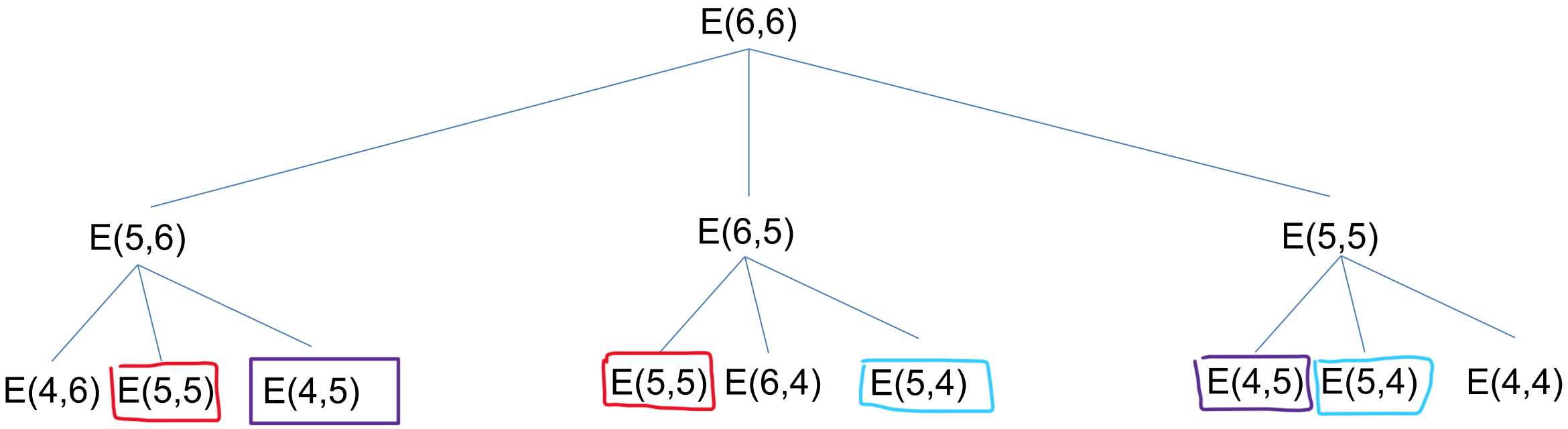


We compute the same subproblem in different branches → **overlapping subproblems**

Edit Distance Problem

Finding **E(n,m)** requires finding all other **E**'s

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$



We compute the same subproblem in different branches → **overlapping subproblems**

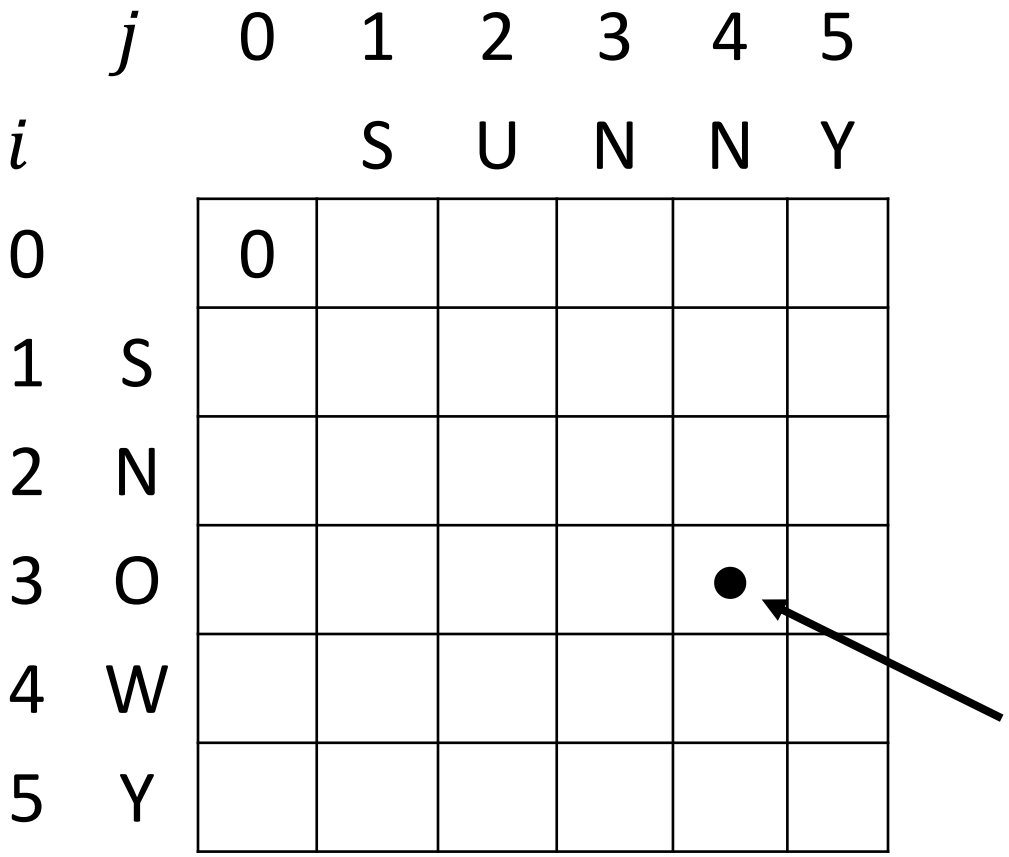
Edit Distance Problem

$$E(i,j) = \min \begin{cases} E(i-1,j) + 1 & x_i \; - \\ E(i,j-1) + 1 & - \; y_j \\ E(i-1,j-1) + \text{diff}(i,j) & x_i \; y_j \end{cases}$$

$$\text{where diff}(i,j) = \begin{cases} 0, & x_i = y_j \\ 1, & x_i \neq y_j \end{cases}$$

Finding $E(n,m)$ requires finding all the other E's, which can be represented in a 2d table with the strings along the axes.

Edit Distance

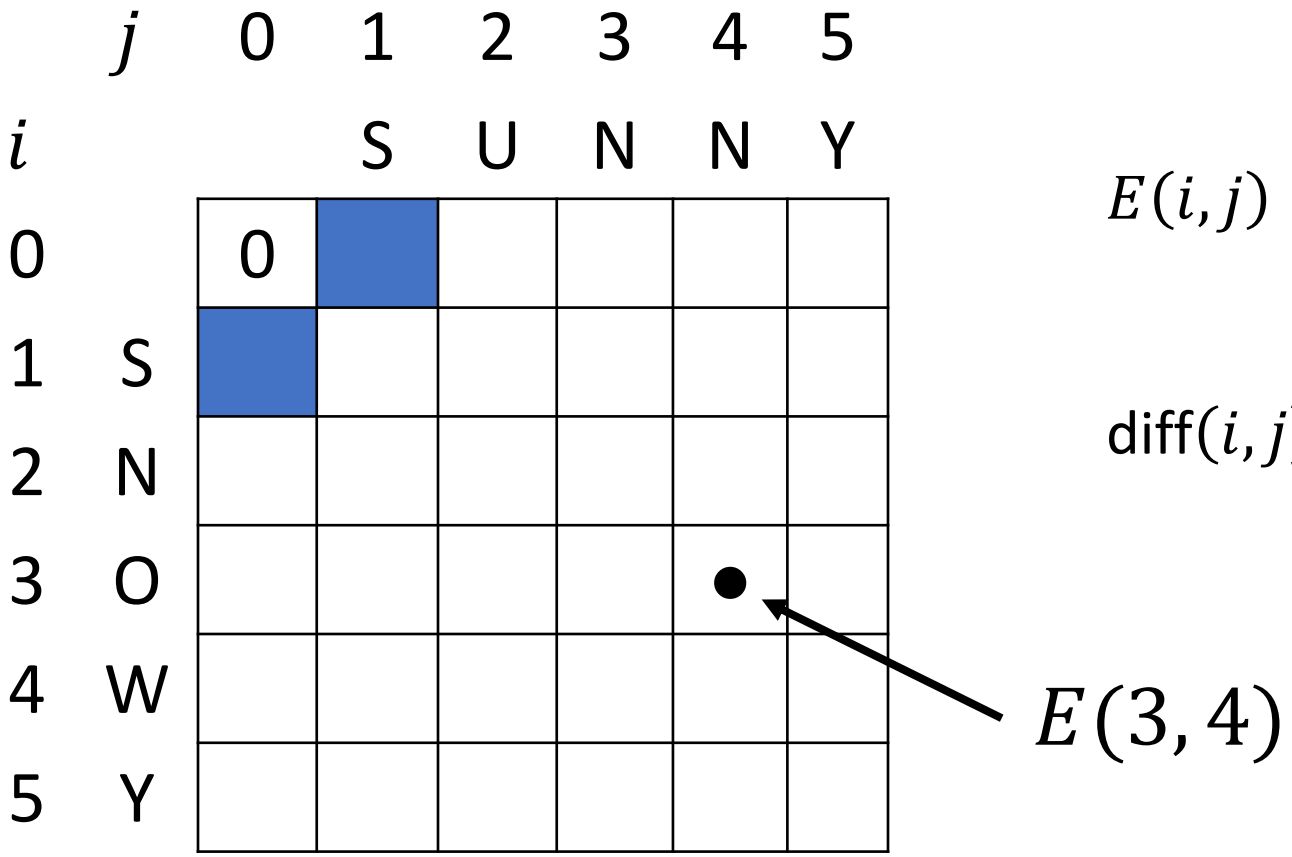


$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & x[i] \neq y[j] \end{cases}$$

Where can we start?

Edit Distance



$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & x[i] \neq y[j] \end{cases}$$

Where can we start?

$E(0,1)$ or $E(1,0)$

Edit Distance

j		0	1	2	3	4	5
i			S	U	N	N	Y
0		0					
1	S						
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(0,1) = \min \begin{cases} E(-1,1)+1 \\ E(0,0)+1 \\ E(-1,0)+1 \end{cases} = ?$$

Edit Distance

	j	0	1	2	3	4	5
i			S	U	N	N	Y
0		0					
1	S						
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(0,1) = \min \begin{cases} \sout{E(-1,1)+1} \\ E(0,0)+1 \\ \sout{E(-1,0)+1} \end{cases} = ?$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1				
1	S						
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(0,1) = \min \begin{cases} \cancel{E(-1,1)+1} \\ E(0,0)+1 \\ \cancel{E(-1,0)+1} \end{cases} = 1$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1				
1	S						
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(1,1) = \min \begin{cases} E(0,1)+1 \\ E(1,0)+1 = ? \\ E(0,0)+0 \end{cases}$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1				
1	S						
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

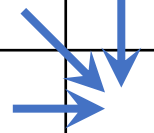
$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(1,1) = \min \begin{cases} E(0,1)+1 \\ \boldsymbol{E(1,0)+1} =? \\ E(0,0)+0 \end{cases}$$

Not calculated yet!

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1				
1	S						
2	N						
3	O						
4	W						
5	Y						



$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

Need upper left hand corner filled out before we can progress.

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2			
1	S						
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(0,2) = \min \begin{cases} E(-1,2)+1 \\ E(0,1)+1 \\ E(-1,1)+1 \end{cases} = 2$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2			
1	S	1					
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(1,0) = \min \begin{cases} E(0,0)+1 \\ E(1,-1)+1 = 1 \\ E(0,-1)+1 \end{cases}$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2			
1	S	1	0				
2	N						
3	O						
4	W						
5	Y						

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

$$E(1,1) = \min \begin{cases} E(0,1)+1 \\ E(1,0)+1 = 0 \\ E(0,0)+0 \end{cases}$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

Running Time?

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

$$\text{diff}(i,j) = \begin{cases} 0, & x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

Running Time?

Fill out $n \times m$ table with constant operations: $O(nm)$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	**3**

Edit distance = **3**.

How can we recreate the actual alignments?

Backtracking.

Ask the question: “How did we get here?”

Edit Distance

	j	0	1	2	3	4	5
i			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	**3**

How did we get to $E(5,5)$?

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4 ←	**3**

How did we get to $E(5,5)$?
From $E(5,4)$?

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4 ←	**3**

How did we get to $E(5,5)$?

From $E(5,4)$? – No. Can never go down in cost.

Edit Distance

i		j 0	1 S	2 U	3 N	4 N	5 Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	**3**

How did we get to $E(5,5)$?

From $E(5,4)$? – No. Can never go down in cost.

From $E(4,5)$?

Edit Distance

	j	0	1	2	3	4	5
i			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	**3**

How did we get to $E(5,5)$?

From $E(5,4)$? – No. Can never go down in cost.

From $E(4,5)$? – No. Need +1 to move that direction.

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

Edit Distance

	j	0	1	2	3	4	5
i			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	**3**

How did we get to $E(5,5)$?

From $E(5,4)$? – No. Can never go down in cost.

From $E(4,5)$? – No. Need +1 to move that direction.

From $E(4,4)$?

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

Edit Distance

	j	0	1	2	3	4	5
i			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	**3**

How did we get to $E(5,5)$?

From $E(5,4)$? – No. Can never go down in cost.

From $E(4,5)$? – No. Need +1 to move that direction.

From $E(4,4)$? – Yes. Match Y's.

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Continuing the process yields all of the optimal solutions.

Diagonal move indicates ?

Vertical move indicates ?

Horizontal move indicates ?

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Continuing the process yields all of the optimal solutions.

Diagonal move indicates match.

Vertical move indicates ?

Horizontal move indicates ?

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Continuing the process yields all of the optimal solutions.

Diagonal move indicates match.

Vertical move indicates space inserted in j.

Horizontal move indicates ?

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

Edit Distance

i		j 0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Continuing the process yields all of the optimal solutions.

Diagonal move indicates match.

Vertical move indicates space inserted in j.

Horizontal move indicates space inserted in i.

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1)+\text{diff}(i,j) \end{cases}$$

Edit Distance

i		j 0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Diagonal move indicates match.

Vertical move indicates space inserted in j.

Horizontal move indicates space inserted in i.

S – N O W Y

S U N N – Y

Edit Distance

	j	0	1	2	3	4	5
i			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Diagonal move indicates match.

Vertical move indicates space inserted in j.

Horizontal move indicates space inserted in i.

Alignment?

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Diagonal move indicates match.

Vertical move indicates space inserted in j.

Horizontal move indicates space inserted in i.

S – N O W Y

S U N – N Y

Edit Distance

i	j	0	1	2	3	4	5
			S	U	N	N	Y
0		0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	N	2	1	1	1	2	3
3	O	3	2	2	2	2	3
4	W	4	3	3	3	3	3
5	Y	5	4	4	4	4	3

Diagonal move indicates match.

Vertical move indicates space inserted in j.

Horizontal move indicates space inserted in i.

S N O W Y

S U N N Y

Applications of Edit Distance

GGAACAGATTGGTCTAATTAGCTTAAGAGAGTAAATTCTGGGATCATTCA

GTAGTAATCACAAATTTACGGTGGGGCTTTTTTGGCGGATCTTTACAGAT

Edit Distance (Levenshtein Distance): 29

Edit Distance is used in computational biology to find how similar two sequences of DNA are

Applications of Edit Distance

We can could use edit distance to correct misspelled words!

`mawntain`

Did you mean **`maintain`** (1) ?
Did you mean **`mountain`** (2) ?
Did you mean **`captain`** (3) ?
Did you mean **`mantis`** (3) ?