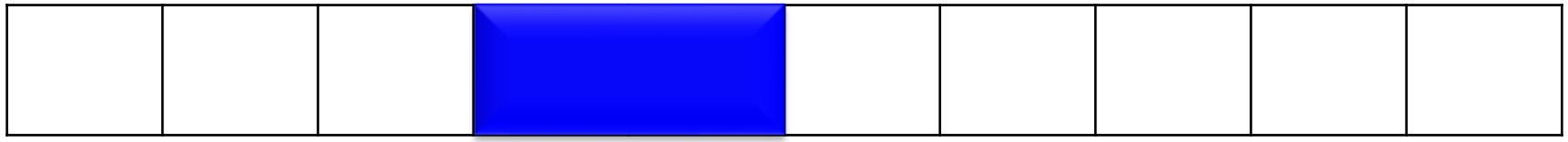
Remember This Problem?

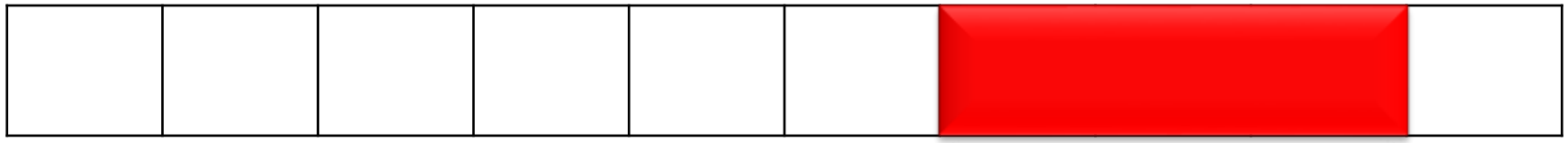
- We have a grid of boxes on a line as shown

Remember This Problem?

- We also have two kinds of tiles
- A blue tile can cover any two consecutive boxes on the grid



- A red tile can cover any three consecutive boxes



Remember This Problem?

- We have unlimited numbers of blue and red tiles
- The goal is to cover the whole grid

Remember This Problem?

Allowed

- Put red tiles next to each other
- Put blue tiles next to each other
- Use only blue tiles
- Use only red tiles

Not Allowed

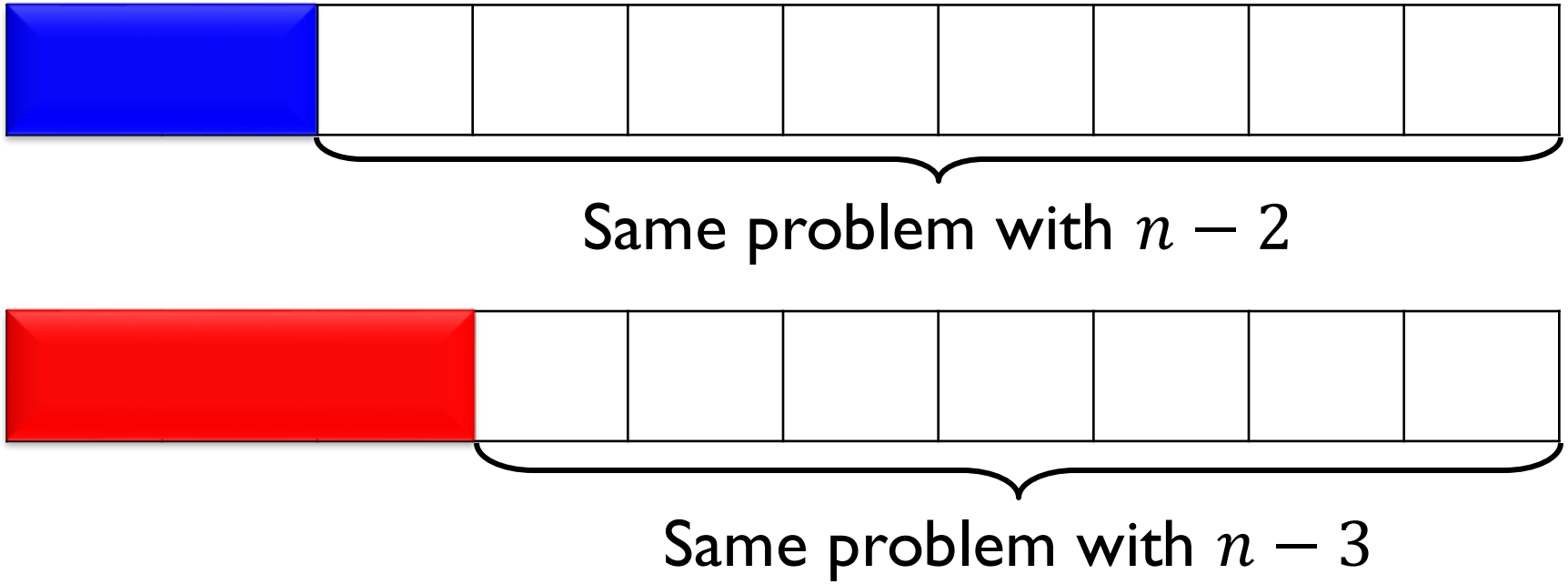
- Tiles not aligned to boxes
- Leaving some box uncovered

Remember This Problem?

- Another way to tile the grid

- Two ways are different if the color of the tile covering a box is different in one way from the other, for at least one box
- How many ways are there to tile a grid of n boxes?

And Its Solution?



And Its Solution?

```
def ways(n):
    if n == 1:
        return 0
    elif n == 2 or n == 3:
        return 1
    else:
        return ways(n - 2) + ways(n - 3)
```

Run It For Bigger and Bigger n

- What is the largest n for which you are willing to wait?
- Why so slow?
  - In the recursion tree, each node spawns two other nodes
  - n levels before reaching base case
  - $O(2^n)$
  - Rough estimate only: the tree is imbalanced – one branch reaches the base case sooner than the other
  - But the actual running time is still *exponential*
- How to make it faster? Let's try computing it ourselves first to see if we can observe anything

How Did We Compute It Faster than the Computer?

- Didn't need to stupidly solve some subproblems
- Because we have *seen it before* and have *memorized* the answer
- Transferring this "smartness" to the computer
  - First time a subproblem is encountered, do it the hard way
  - So that hard work is not wasted, store the answer in memory
  - Every succeeding time, just retrieve and instantly return the answer from memory

Where to Store the Answers?

- A list where the ith item stores the answer for `ways(i)`
- Initially contains `None` to mean "answer was not computed before"
  - In C++, use a dummy value instead, such as -1
    - We know -1 can't be a valid output of `ways`

A Simple Change to Make It Faster

```
memo = [None for n in range(N)]

def ways(n):
    if memo[n] is None:
        if n == 1:
            answer = 0
        elif n == 2 or n == 3:
            answer = 1
        else:
            answer = ways(n - 2) + ways(n - 3)
        memo[n] = answer
        return answer
    else:
        return memo[n]
```

Compute as usual, but store answer in memo before returning

Answer already computed before, just retrieve from memo

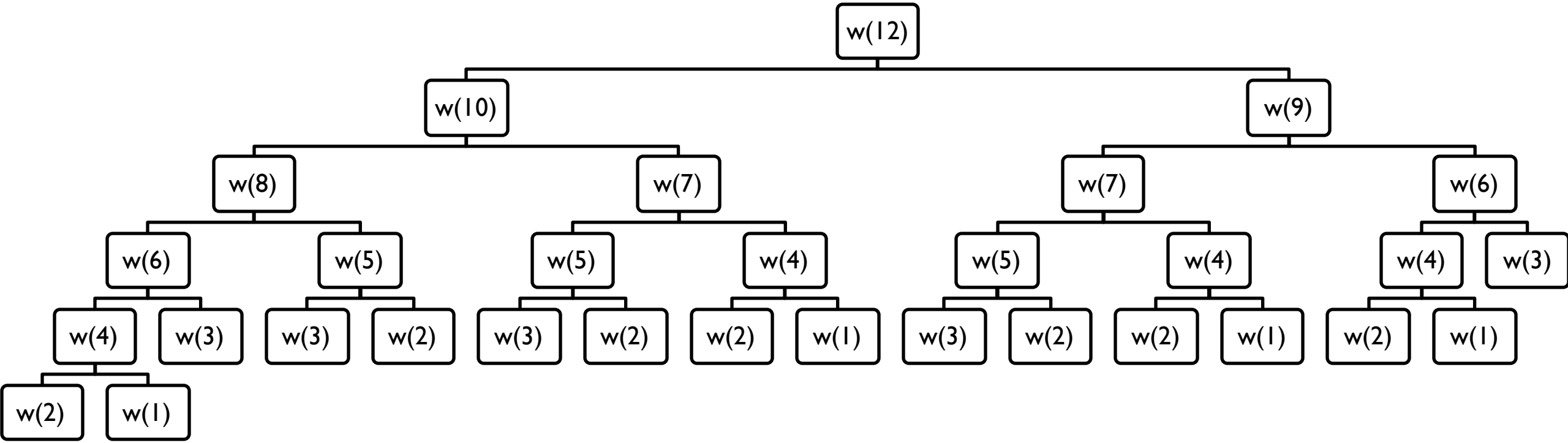
We Can Slightly Simplify

```
memo = [None for n in range(N)]

def ways(n):
    if memo[n] is None:
        if n == 1:
            answer = 0
        elif n == 2 or n == 3:
            answer = 1
        else:
            answer = ways(n - 2) + ways(n - 3)
        memo[n] = answer
    return memo[n]
```

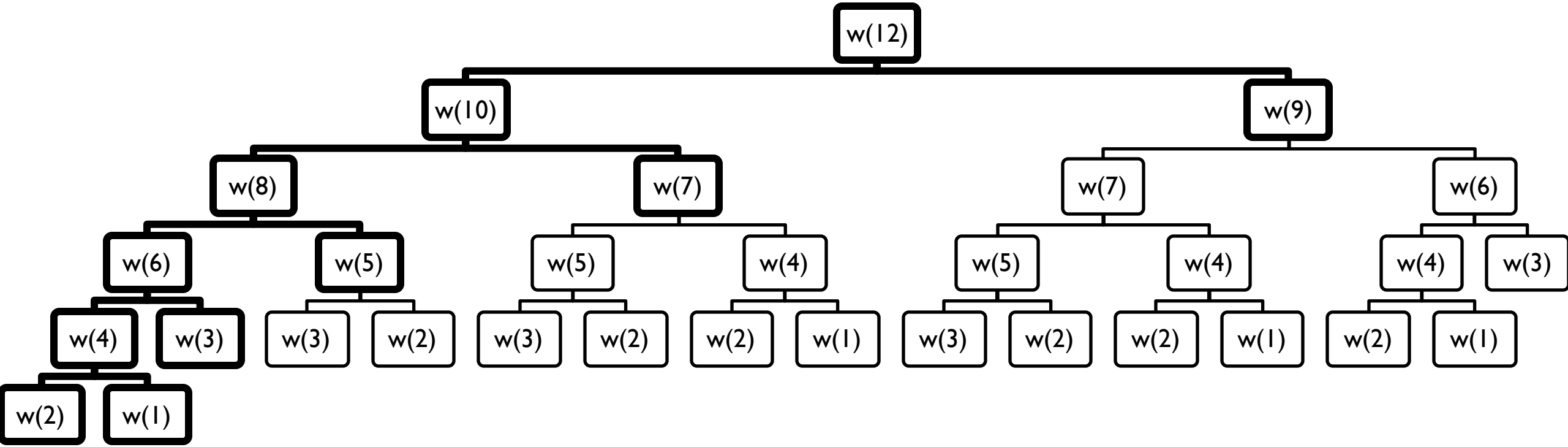
Why Is It Faster? How Fast Is It Exactly?

- Look at the recursion tree



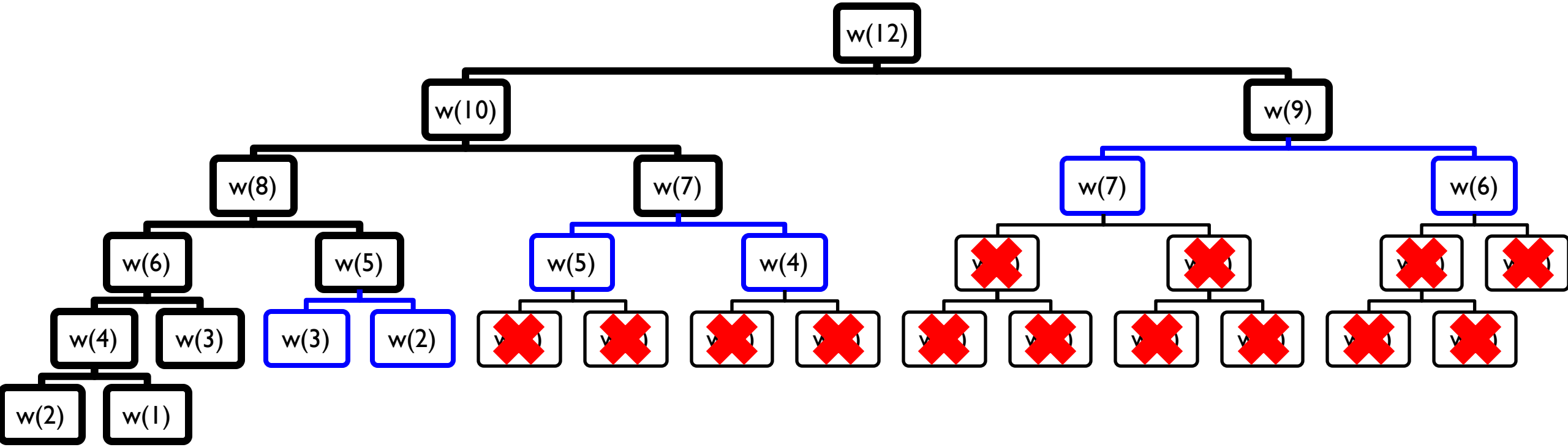
Why Is It Faster? How Fast Is It Exactly?

- First time → solve normally



Why Is It Faster? How Fast Is It Exactly?

- Solved before → further calls get pruned



Can You Figure It Out?

- What is the running time of `ways(n)`, using the improved definition?
- Only black nodes and blue nodes contribute
- Black nodes → first encounters
- Blue nodes → at most two for each black node
- $O(1)$ non-recursive work in each node
- Running time is directly proportional to number of first encounters, or number of unique subproblems
- $\Theta(n)$

This Totally Works for Any Recursive Function

```
memo = [None for i in range(MAX_PROBLEM_SIZE)]

def f(subproblem):
    if memo[subproblem] is None:
        # Solve normally, with recursion,
        # but instead of returning the answer,
        # assign it to "answer"
        memo[subproblem] = answer
    return memo[subproblem]
```

As Long As It's Pure

- **Pure** function: always returns the same value when called with the same values as input

As Long As It's Pure

- Example of impure function:

```
a = [1, 2, 3]
def impure(n):
    a[n] += 1
    return a[n]
```

- Why it's impure:

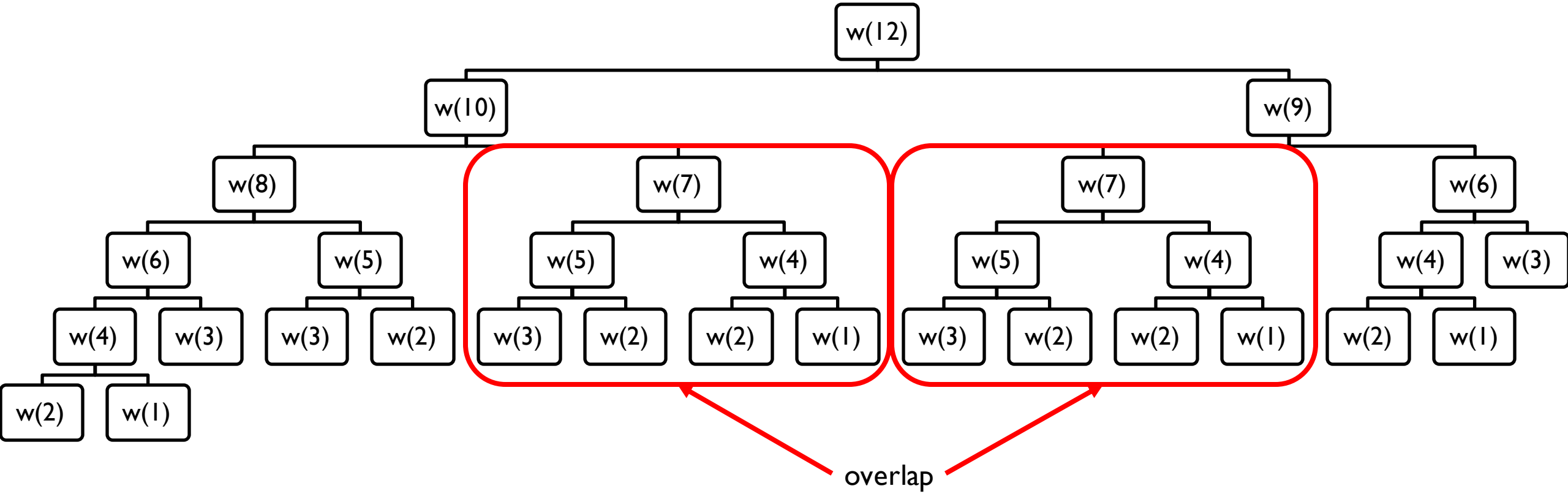
```
print(impure(2))
print(impure(2))
print(impure(2))
```

- Prints three different values

As Long As It's Pure

- **Pure** function: always returns the same value when called with the same values as input
- Think: why can't we apply this `memo` technique on impure functions?
- If we expect answers to change at each call, storing old answers is useless because they would be wrong by the time they are retrieved

We Only Gain Efficiency If There Are Overlapping Subproblems

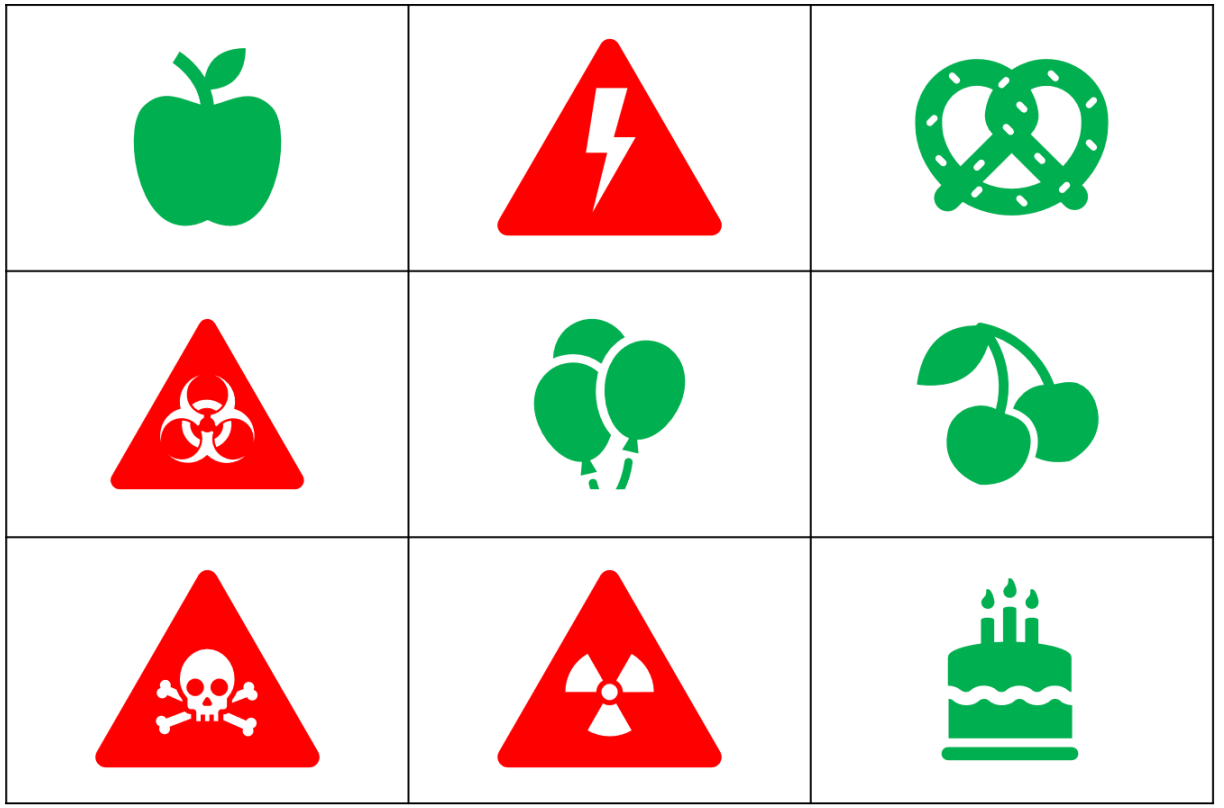


People Call This Technique Memoization

- Looks like a misspelling but it is what it is
  - Idea: to memoize = to put into a memo
- "Memorization" would have been ok, but we're stuck with "memoization" now so that's what we'll use
- Key idea: if you memo(r)ize answers, you can (sometimes) compute faster

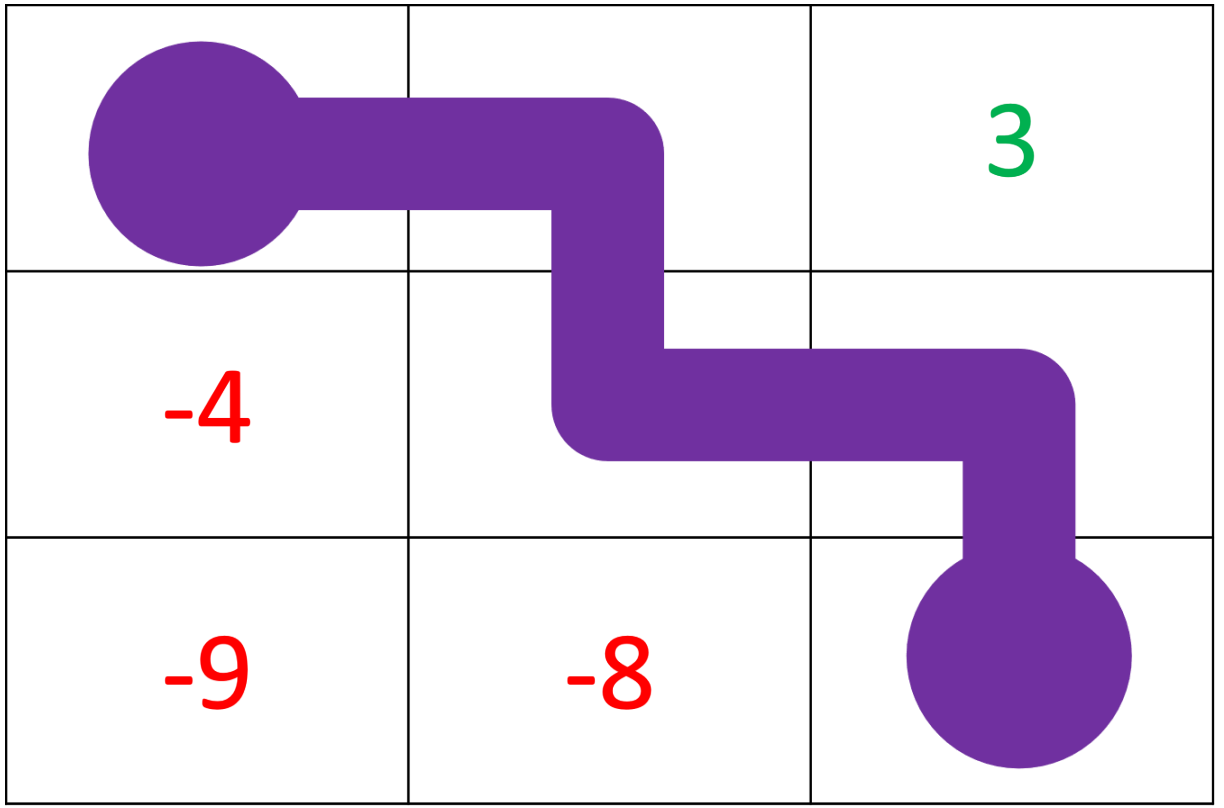
A Robot is Standing in a Maze

- *Health tiles* increase points
- *Harm tiles* decrease points



Represent the Maze as a Grid of Integers

- Points increase/decrease by specific amounts
- Start at the upper-left corner
  - Automatically gain/lose points on that tile
- End at lower-right corner
  - Automatically gain/lose points on that tile
- Can only move downwards or rightwards
  - Gain/lose points on all stepped tiles
- What is the best path?



Solving in General

- A list `A` of n lists containing m numbers each
- `A[i][j]` is points gained/lost when stepping on ith row and jth column of the grid

A Smart Idea

- At every step, pick the direction with the larger number

At $(0,0)$: $-1 > -4$ so go right

At $(0,1)$: $5 > 3$ so go down

At $(1,1)$: $2 > -8$ so go right

2	-1	3
-4	5	2
-9	-8	7

But It's Wrong!

- Can try other smart ideas but you'll find problems with them
  - Example: force pass through largest value in the grid
    - What if there are multiple and you can't pass through all of them? Which ones to pick?
- Instead, think *complete search*
- Try all possible paths! Surely one of them is the right one

1	1	1
-1	-1	1
999	-1	1

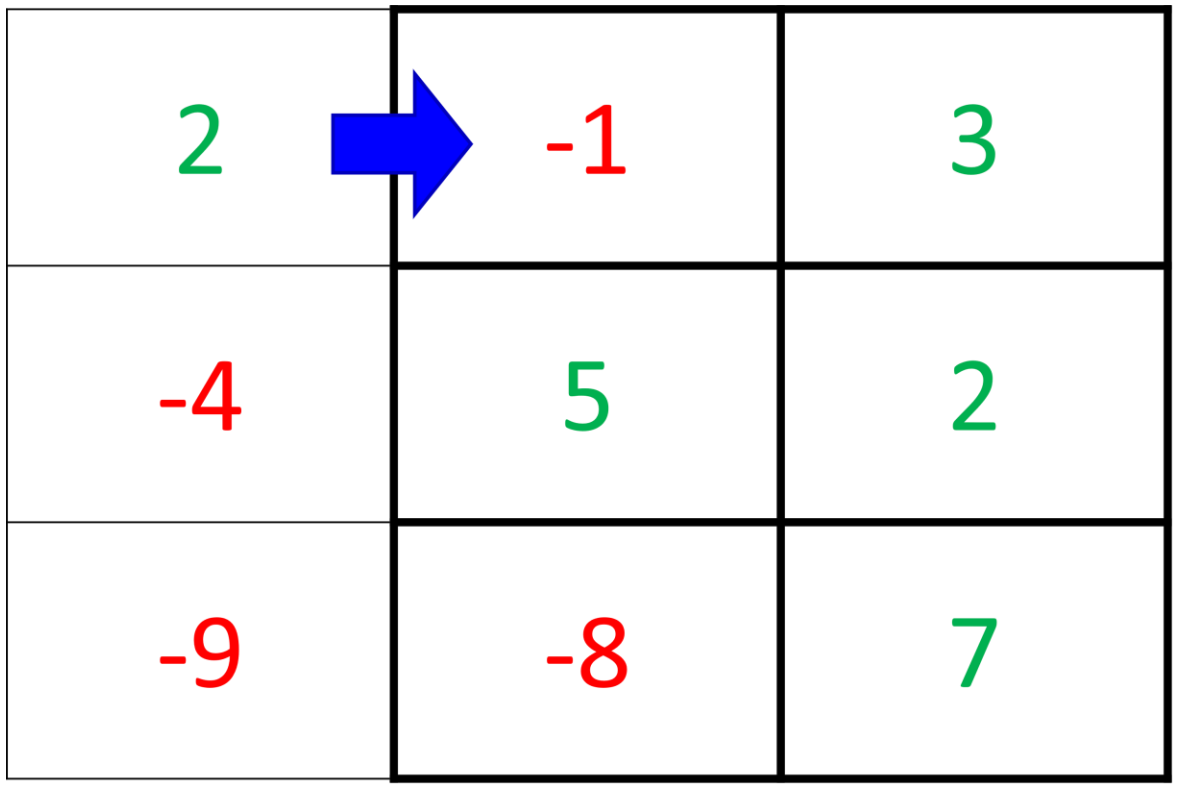
Enumerating All Possible Paths

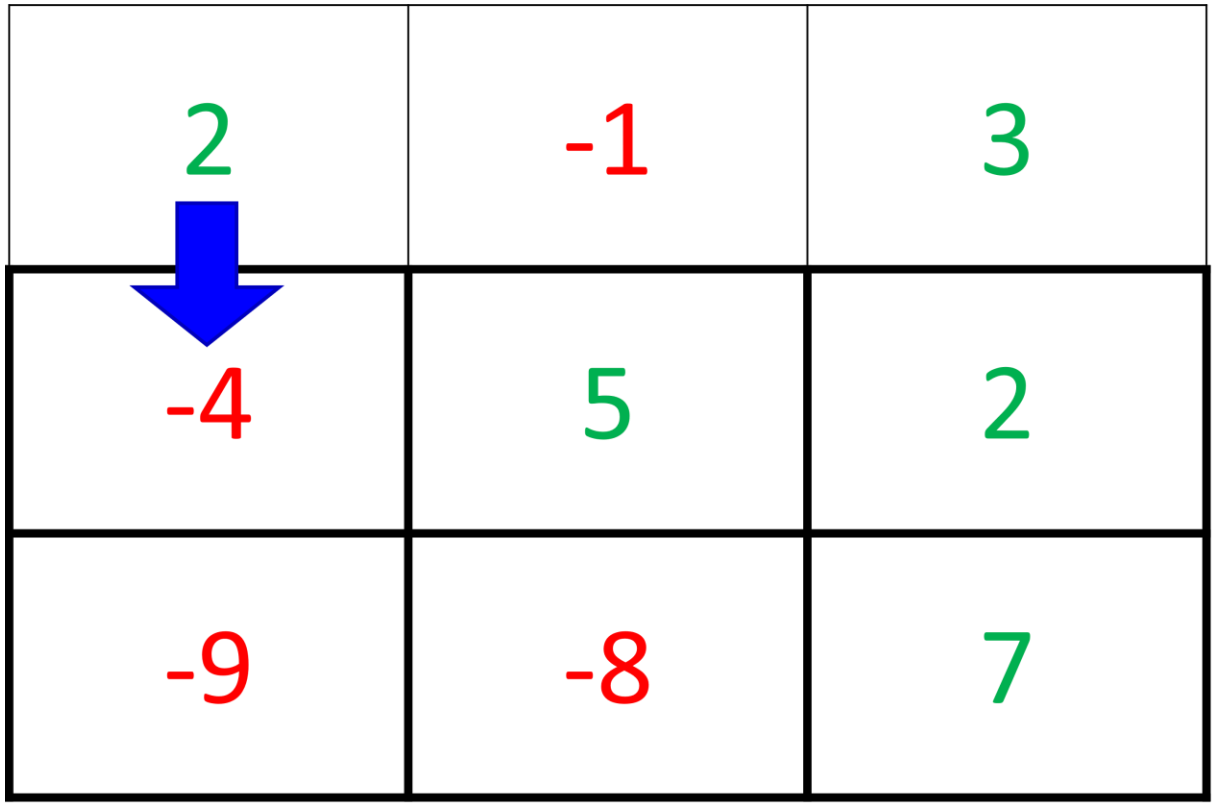
- Paths are not nicely given to us in a list, need to think *recursive backtracking*
- Break down "decide a path" into stages: "decide right or down" $n + m - 2$ times
  - Choices at each stage are basically the same
  - Except: at rightmost column or lowest row

Enumerating All Possible Paths

- Paths are not nicely given to us in a list, need to think *recursive backtracking*
- We can also think about the recursive structure directly
- Find a single step that transforms the problem into a smaller subproblem which looks like the original
  - Literally, take a step in the grid
  - Notice that the resulting problem looks like the original – find the best path in the part of the grid without either the first row or the first column

Recursive Structure

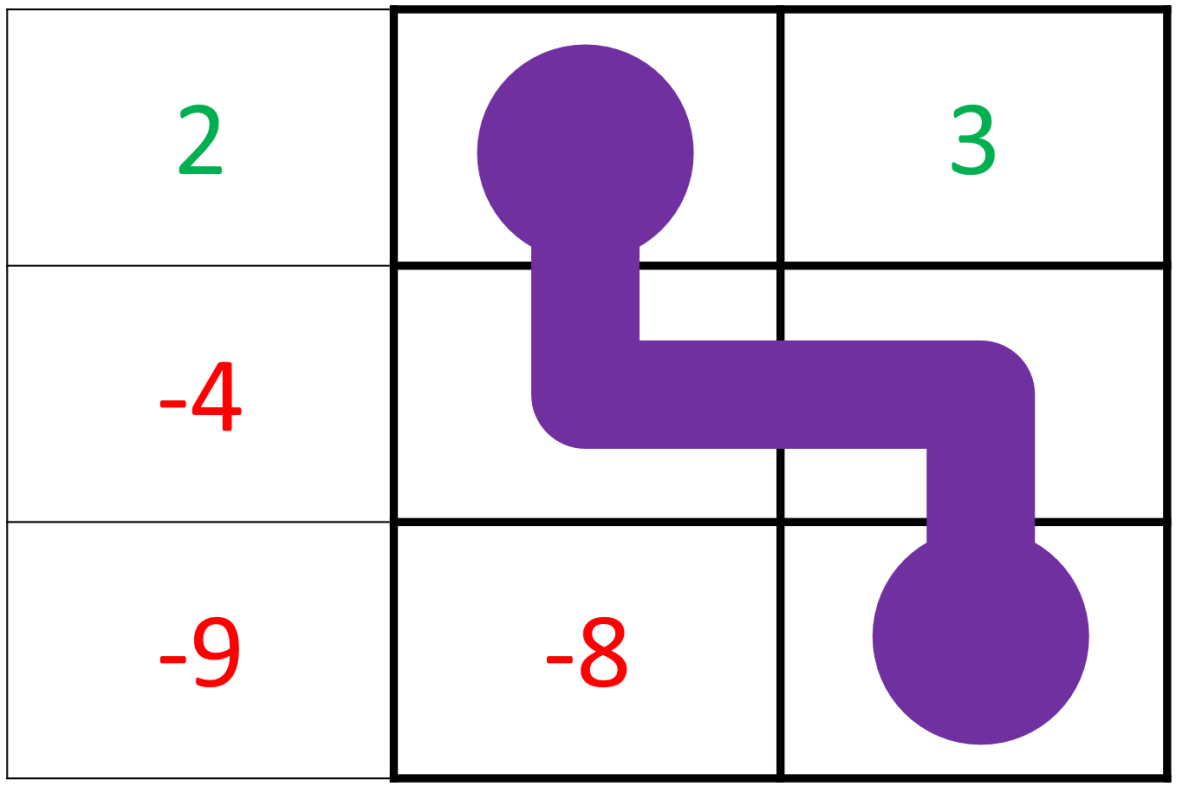


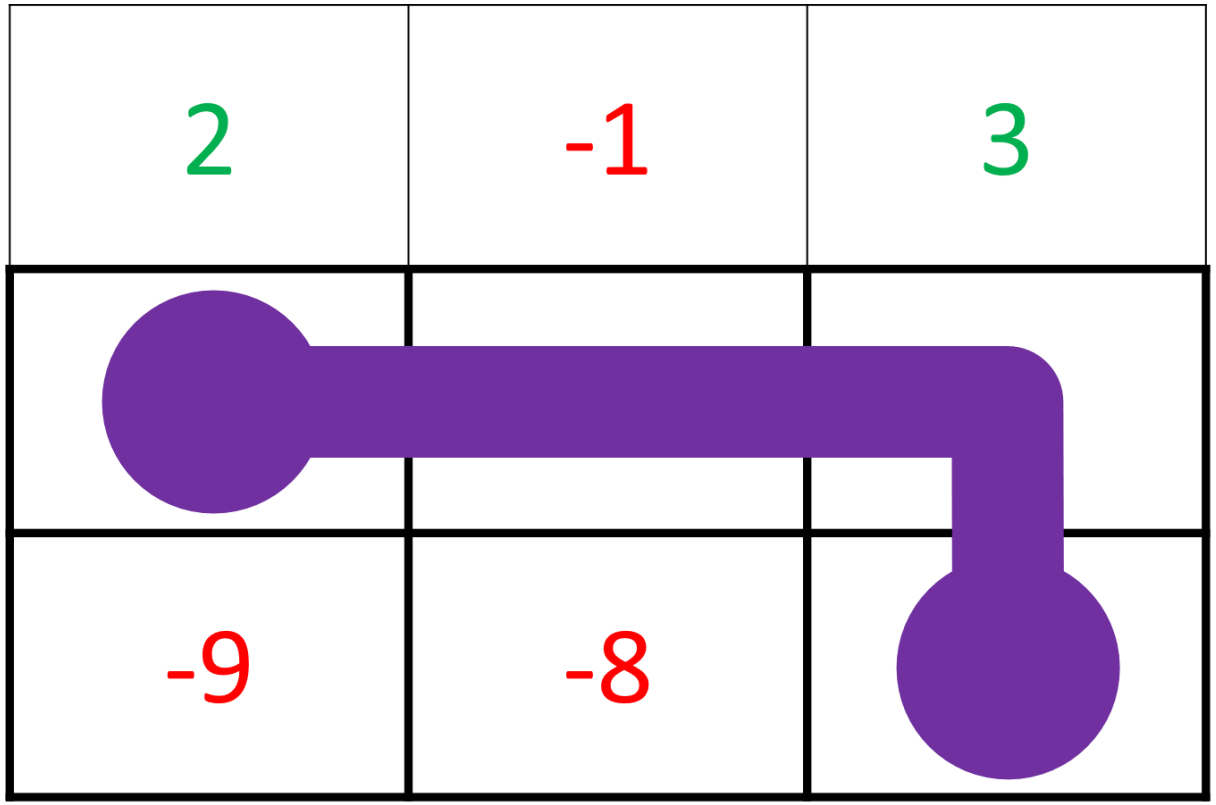


Recursive Structure

- **Believe** that there is a way to get the best path after taking the first step

Recursive Structure





Recursive Structure

- The best path for the whole grid must contain one of these two options
  - Otherwise, we can replace the not-best partial path with one of these two, and get a better overall path

Recursive Structure

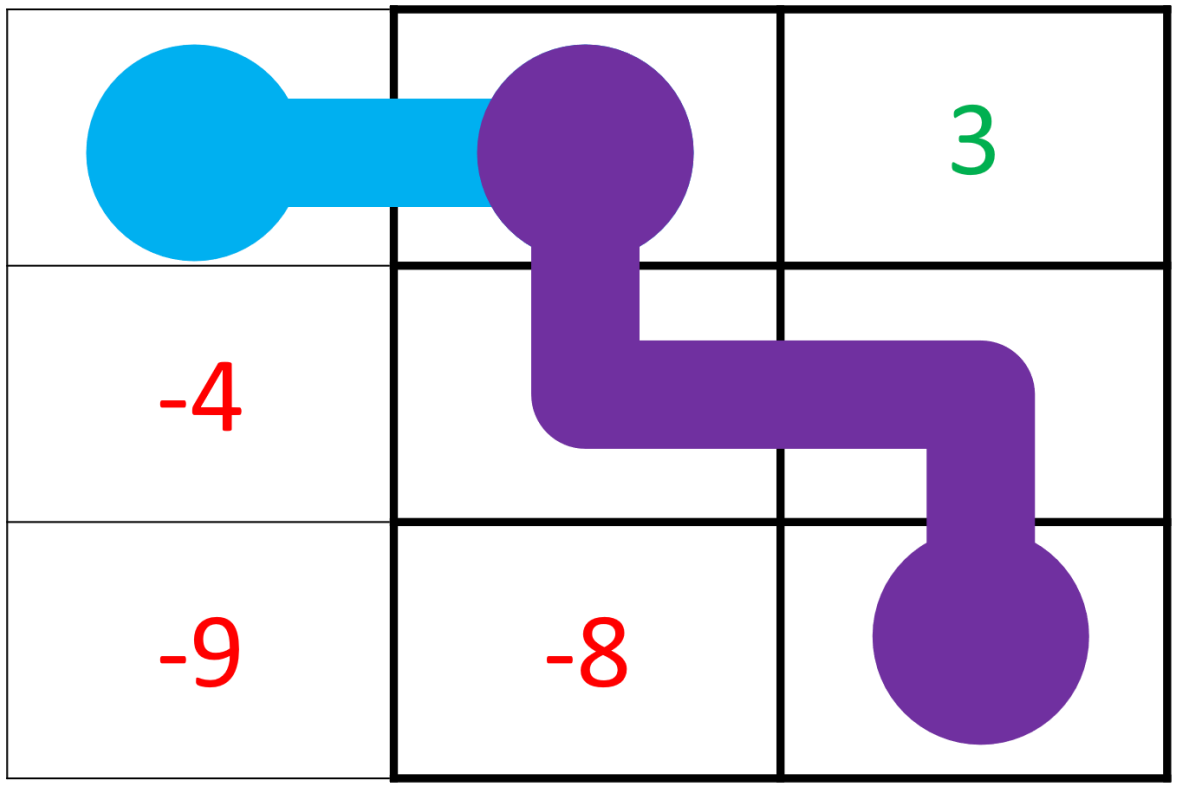
- We don't know which step is the correct first step, so just try both

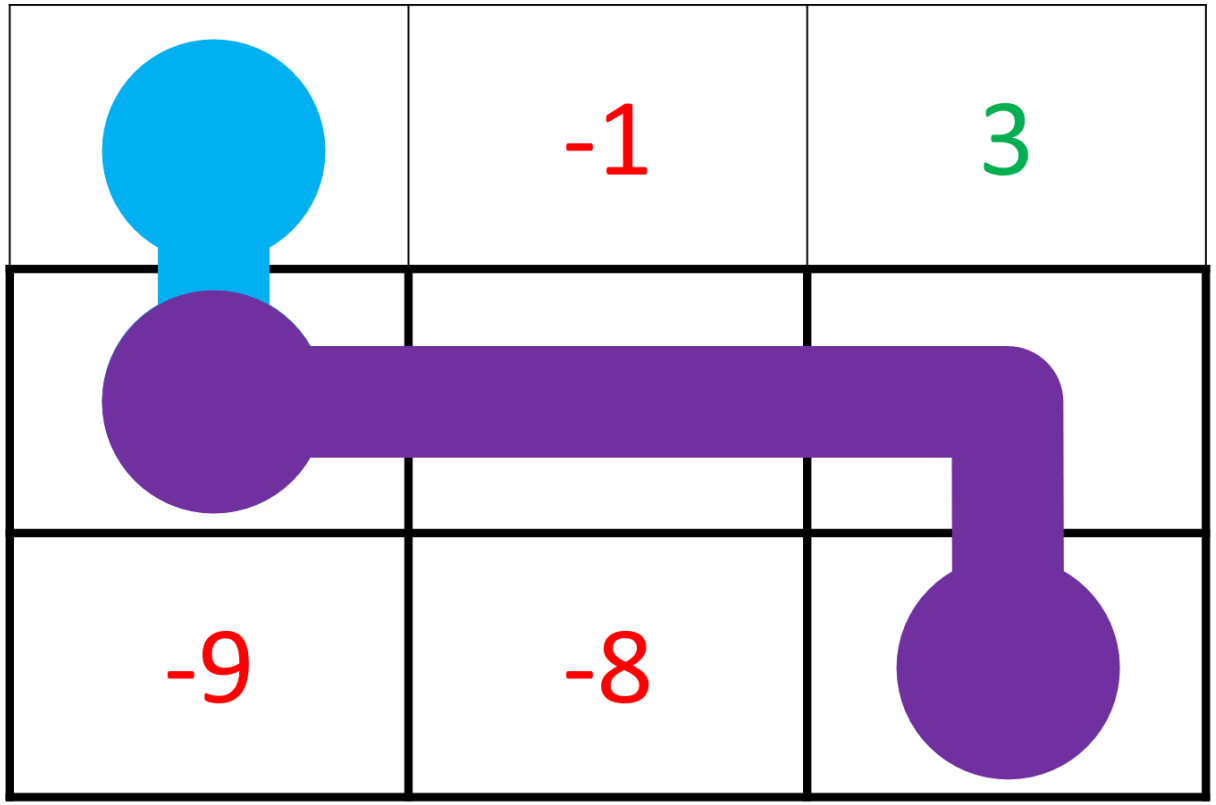
```
best path = max(
    best path that starts with a rightward step,
    best path that starts with a downward step
)
```

Recursive Structure

- Try one option = solve with recursion, then just transform the answer obtained from the subproblem into the answer for the original problem
  - In short, add the tile currently stepped on

Recursive Structure





Recursive Structure

```
best path = max(
    tile stepped now + best path after going right,
    tile stepped now + best path after going down
)
```

- Or, more simply

```
best path = tile stepped now + max(
    best path after going right,
    best path after going down
)
```

Translating into Code

- First, let's not think of outputting the best path, just the maximum points obtained

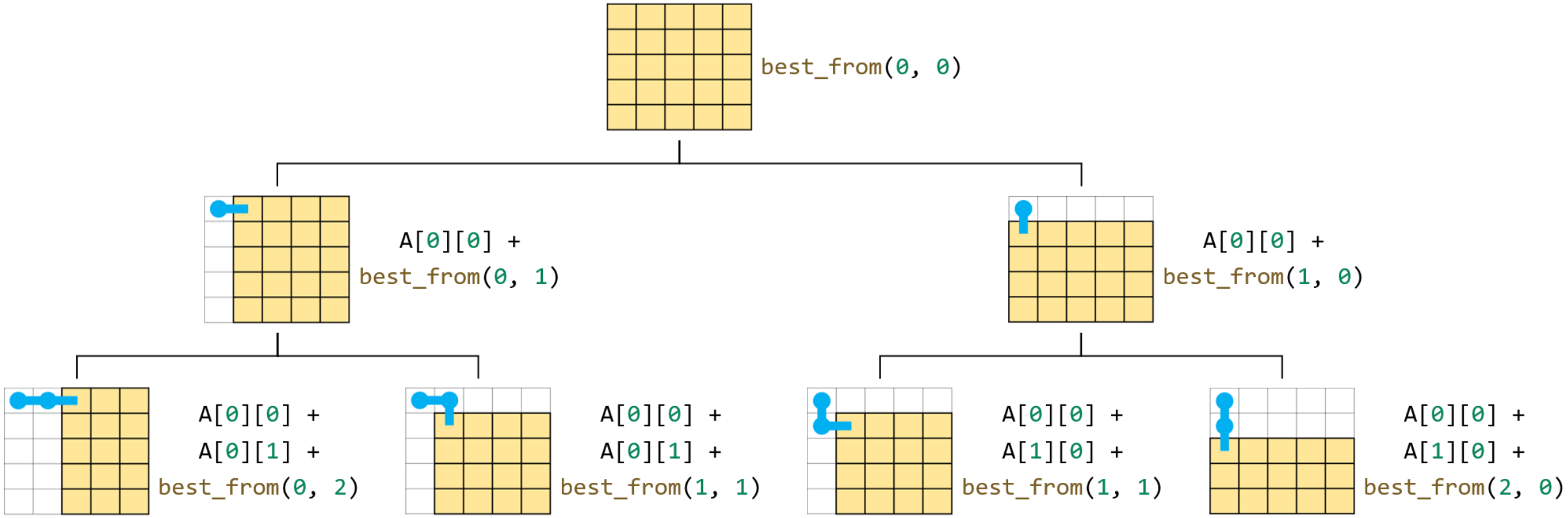
Translating into Code

- What variables do we need to identify a subproblem?
  - Which subgrid is considered
    - The upper-left corner
      - Which row, which column
    - The lower-right corner
      - It's the same for all subproblems, so don't need to include as a parameter
  - For same reason, the grid itself, number of rows, and number of columns also not included
  - Notice: upper-left corner = "current position"

Translating into Code

```
def best_from(row, col):
    return A[row][col] + max(
        best_from(row, col + 1),
        best_from(row + 1, col)
    )
```

If you're not convinced this really tries all paths, stare at this diagram for a few minutes



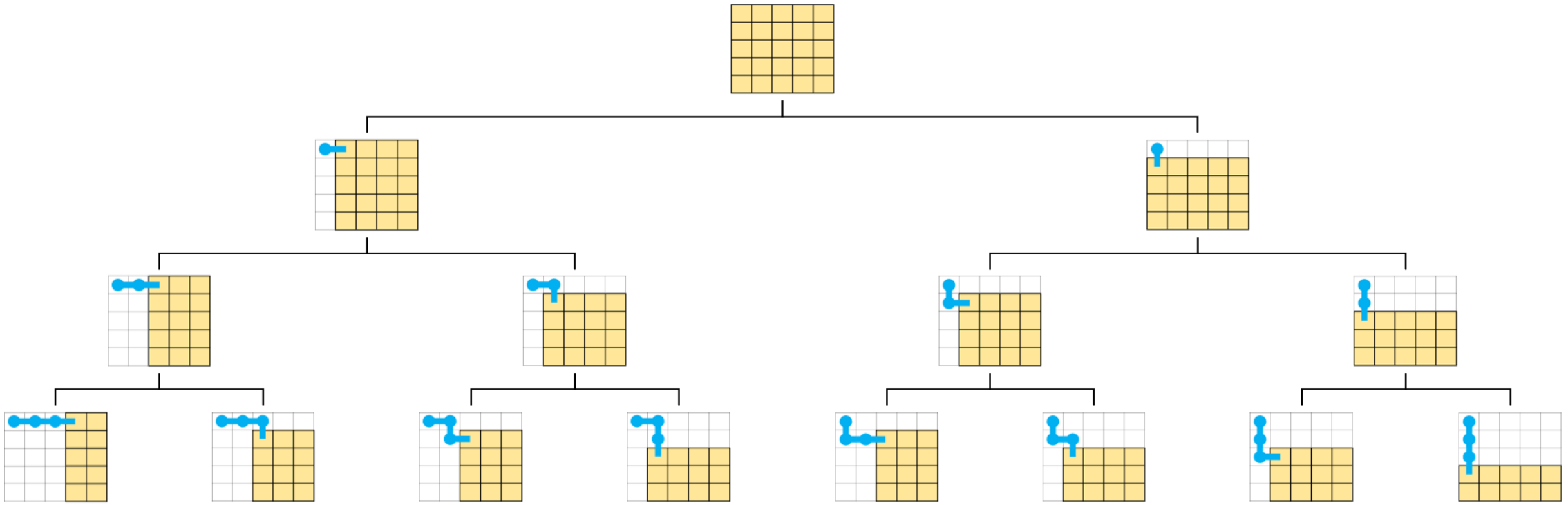
Now with (Literal) Corner and Edge Cases

```
def best_from(row, col):
    if row == n - 1 and col == m - 1:
        return A[row][col]
    elif row == n - 1: # and col != m - 1
        return A[row][col] + best_from(row, col + 1)
    elif col == m - 1: # and row != n - 1
        return A[row][col] + best_from(row + 1, col)
    else: # col != m - 1 and row != n - 1
        return A[row][col] + max(
            best_from(row, col + 1),
            best_from(row + 1, col)
        )
```

Is This Solution Efficient or Not?

- For almost all subproblems, recursion branches into 2 at each step
- $\Theta(n + m)$ steps
- $\Theta(2^{n+m})$
- For a grid with 50 rows and 50 columns, this is already 1000 times the age of the universe (13.4 billion years)
- Any suggestions to make it faster?

Where Are the Overlapping Subproblems?



So, We Can Memoize!

```
memo = [[None for col in range(m + 1)] for row in range(n + 1)]

def best_from(row, col):
    if memo[row][col] is None:
        if row == n - 1 and col == m - 1:
            answer = A[row][col]
        elif row == n - 1: # and col != m - 1
            answer = A[row][col] + best_from(row, col + 1)
        elif col == m - 1: # and row != n - 1
            answer = A[row][col] + best_from(row + 1, col)
        else: # col != m - 1 and row != n - 1
            answer = A[row][col] + max(
                best_from(row, col + 1),
                best_from(row + 1, col)
            )
        memo[row][col] = answer
    return memo[row][col]
```

So, We Can Memoize!

- Again, notice how the main bulk of the code is the same as before, the memoization parts just added somewhat "mechanically"
- This time, since the function has two parameters, we use a list of lists to have a proper place in the memo for each subproblem

Note for C++ Programmers

- The grid has negative values, so -1 could be a legitimate answer to a subproblem
- Cannot use -1 as a dummy value
  - Otherwise, might mistakenly think subproblem has not been solved yet and recompute manually, losing efficiency
- Instead, use something like

```
n * m * MAX_VALUE_IN_GRID + 1
```

With Memoization, How Fast Is It Exactly?

- n possible values of `row` and m possible values of `col`, so $n \times m$ unique subproblems
- Each subproblem solved manually once
  - Non-recursive work is $\Theta(1)$
- Recursive call to already solved subproblem takes $\Theta(1)$ to retrieve the answer
- Recursive call to not yet solved subproblem already counted in "each subproblem solved manually once"
- $\Theta(nm)$

With Memoization, How Fast Is It Exactly?

- From this and the tiling puzzle, seems there's a shortcut:

Running time = Θ(number of unique subproblems)
(black nodes in the recursion tree)

With Memoization, How Fast Is It Exactly?

- $\Theta(nm)$ is such a big improvement from $\Theta(2^{n+m})$!
- On grid with 50 rows and 50 columns, one finishes instantly, the other takes several universe lifetimes
- And we applied only a simple trick!
- Not a lot of problem-specific smartness

Printing the Actual Path

- We can modify the function to return the path, but this leads to uglier and slower code
  - Memo will contain lists instead of numbers
- Vaguely feels like shortest paths with BFS
  - What we did there: reconstruct the paths in a separate phase after computing the distances
  - Calling `best_from(0, 0)` will make `memo[i][j]` store the best "distance" to the goal from every position (i, j)

We Can Do Something Similar Here

```
path = [[0, 0]]
while path[-1] != [n - 1, m - 1]:
  row, col = path[-1]
  if row == n - 1:
    path.append([row, col + 1])
  elif col == m - 1:
    path.append([row + 1, col])
  else:
    if memo[row + 1][col] > memo[row][col + 1]:
      path.append([row + 1, col])
    else:
      path.append([row, col + 1])
```

We Can Do Something Similar Here

- There are only $\Theta(n + m)$ steps in the path
- Each step is only $\Theta(1)$ for accessing the memo
- The bottleneck is still in filling up the memo, $\Theta(nm)$

Actually, We Can Use `best_from` Directly, Even Without Calling `best_from(0, 0)`

- And simplify the logic a bit…

```
path = [[0, 0]]
while path[-1] != [n - 1, m - 1]:
  row, col = path[-1]
  if col == m - 1 or best_from(row + 1, col) > best_from(row, col + 1):
    path.append([row + 1, col])
  else:
    path.append([row, col + 1])
```

- Think about why this works (and still in $\Theta(nm)$ time)

Making Change

Making Change

- You buy from a vending machine and expect to get P pesos change
- It can always return P 1-peso coins, but that's annoying
- Natural goal: minimize number of pieces of money to give out to make P
- Easy version for now: machine has either an infinite number or none of each denomination
  - Problem turns out to be way harder with finite non-zero of each denomination

Turning This into a Programming Problem

- Input:
  - An integer amount P
  - A list of denominations which are available in infinite number
    - Those not in the list have run out
    - To make it simpler, assume ₱1 is always in this list
      - Avoid cases where making P is impossible
- Output:
  - List of bills or coins to give out
  - Can start with simpler version: just the minimum number of pieces needed

A Simple Algorithm

- Repeatedly give out the largest denomination less than or equal to the amount needed
- Example:
  - ₱1234 → give out ₱1000 → ₱234 remains
  - ₱234 → give out ₱200 → ₱34 remains
  - ₱34 → give out ₱20 → ₱14 remains
  - ₱14 → give out ₱10 → ₱4 remains
  - ₱4 → give out ₱1 (repeat 4 times)
- This is what cashiers normally do when they give change

The Cashier's Algorithm

```
pieces_of_money_used = 0
while amount > 0:
    amount -= max(d for d in denominations if d <= amount)
    pieces_of_money_used += 1
return pieces_of_money_used
```

- Does it work correctly for all cases?
- Can you come up with an amount and a set of denominations for which this algorithm gives a wrong answer?

The Cashier's Algorithm Is Wrong!

- Denominations: $[1, 20, 50]$
- Amount: 60
- Cashier's algorithm gives out one ₱50 and ten ₱1
- Much better answer: three ₱20

The Cashier's Algorithm Is Wrong!

- Comes from the following intuitive idea: "to minimize the number of pieces used, reduce the amount by as much as possible with every single piece."
- Sounds reasonable, but intuition isn't always right
- Being clever is not bad – cleverness can lead to faster algorithms and is sometimes required
- However, it's much harder to guarantee correct
- Also, the problem must allow it
  - For some problems, no amount of cleverness is going to help

The Cashier's Algorithm Is Wrong!

- On the other hand, complete search is *always* right
- And works for *all* problems
- Unfortunately, it's slow

Dynamic Programming to the Rescue!

- **Dynamic programming**: do (careful) complete search and then memoize
  - Name has no meaning – person who invented it just wanted a cool-sounding name for the politicians funding his research
- Guaranteed correct because it's just complete search
- But fast because of memoization!

The Complete Search Part is Typically Recursive, and the Thought Process is Always Like This

1. Think about one step that relates a problem to a smaller subproblem
2. Believe the smaller subproblem has already been solved
3. Use the answer to the smaller subproblem to construct the answer to the original problem, applying some operation related to step 1
4. With more than one choice for a step, try all choices "in parallel" and take a sum/min/max over all these choices depending on the problem

The Recursion Suggests Itself If You Imagine the Solution Construction as a Step-by-Step Process

- A step in this step-by-step process is usually the natural step to use to bridge a subproblem to smaller subproblems
- Example:
  - making change = giving out bills or coins one at a time
  - relate original problem P to subproblem that results *after* giving out one piece with value d, in other words subproblem $P - d$
  - $P - d$ must be made in the best way, so that P is made in the best way
    - This is what allows recursion to work
    - If this were not true, recursion would be wrong

Making Change Recursively

- First pretend available denominations are always $[1, 2, 5]$
- The best solution must be one of the following:
  - Give out a ₱1 coin and recursively obtain the best solution for the remaining amount
  - Give out a ₱2 coin and recursively obtain the best solution for the remaining amount
  - Give out a ₱5 coin and recursively obtain the best solution for the remaining amount
- We don't know which one, so try all of them and take the one which results in the fewest pieces

Translating into Code: Basic Idea / Recursive Case

- When making a single step, the remaining amount decreases by the value of the coin given out, and the number of coins increases by one

```
def min_to_make(amount):
    return 1 + min_to_make(amount - value_given_out)
```

Translating into Code: Basic Idea / Recursive Case

- Can give out ₱1, ₱2, or ₱5

```
def min_to_make(amount):
    1 + min_to_make(amount - 1)
    1 + min_to_make(amount - 2)
    1 + min_to_make(amount - 5)
```

Translating into Code: Basic Idea / Recursive Case

- And we want to minimize

```
def min_to_make(amount):
    return min(
        1 + min_to_make(amount - 1),
        1 + min_to_make(amount - 2),
        1 + min_to_make(amount - 5)
    )
```

Translating into Code: Details / Base Case

- When amount is 0, we're done, so 0 coins needed for that case

```
def min_to_make(amount):
    if amount > 0:
        return min(
            1 + min_to_make(amount - 1),
            1 + min_to_make(amount - 2),
            1 + min_to_make(amount - 5)
        )
    else:
        return 0
```

Translating into Code: Details / Base Case

- Can only give out a coin if its value is less than or equal the remaining amount
- Many different ways to implement this check
- Simple way: if some choice is invalid, make it ∞

Translating into Code: Details / Base Case

```
import math

def min_to_make(amount):
  if amount > 0:
    return min(
      1 + min_to_make(amount - 1),
      1 + min_to_make(amount - 2) if 2 <= amount else math.inf,
      1 + min_to_make(amount - 5) if 5 <= amount else math.inf
    )
  else:
    return 0
```

Translating into Code: Details / Base Case

- Can only give out a coin if its value is less than or equal the remaining amount
- Many different ways to implement this check
- Simple way: if some choice is invalid, make it ∞
  - Why this works: $\min(x, \infty) = x$
  - So, branch(es) with ∞ are just ignored
  - ₱1 is always a valid choice, so can never return ∞

Generalizing to Any Set of Denominations

- Replace the individually specified branches with a loop

```
def min_to_make(amount):
  if amount > 0:
    return min(
      1 + min_to_make(amount - d) if d <= amount else math.inf
      for d in denominations
    )
  else:
    return 0
```

Generalizing to Any Set of Denominations

- Can simplify a bit (just *filter* out the invalid choices)

```
def min_to_make(amount):
  if amount > 0:
    return min(
      1 + min_to_make(amount - d)
      for d in denominations
      if d <= amount
    )
  else:
    return 0
```

In C++

```
int min_to_make(int amount) {
  if(amount > 0) {
    int best = INFINITY_OR_A_REALLY_LARGE_NUMBER;
    for(int d : denominations)
      if(d <= amount)
          best = min(best, 1 + min_to_make(amount - d));
    return best;
  } else {
    return 0;
  }
}
```

Exercise

- Draw the recursion tree for `min_to_make(10)` to convince yourself that it indeed generates all the ways to make change for `10`
- You can probably guess what's coming next…
- Do you notice any overlapping subproblems?

Without Memoization, How Slow Is It?

- Let D be the number of denominations and P be the amount given in the input
- Roughly speaking, each step multiplies the number of branches by D
- Roughly speaking, P steps required to reach the base case
- Very roughly, $O(D^P)$
- This is very imprecise – actual running time is better
- But point is, it's still exponential so it's painfully slow
- Just run the un-memoized function if you don't believe

So Memoize, Exactly Like Before

```
memo = [None for amount in range(p + 1)]

def min_to_make(amount):
    if memo[amount] is None:
        if amount > 0:
            answer = min(
                1 + min_to_make(amount - d)
                for d in denominations
                if d <= amount
            )
        else:
            answer = 0
        memo[amount] = answer
    return memo[amount]

print(min_to_make(p))
```

In C++

```
vector<int> memo;
int min_to_make(int amount) {
  if(memo[amount] == -1) {
    int answer;
    if(amount > 0) {
      int best = INFINITY;
      for(int d : denominations)
        if(d <= amount)
          best = min(
            best,
            1 + min_to_make(amount - d)
          );
      answer = best;
    } else {
      answer = 0;
    }
    memo[amount] = answer;
  }
  return memo[amount];
}
```

```
int main() {
  ...
  memo.resize(p, -1);
  cout << min_to_make(p) << endl;
}
```

With Memoization, How Fast Is It?

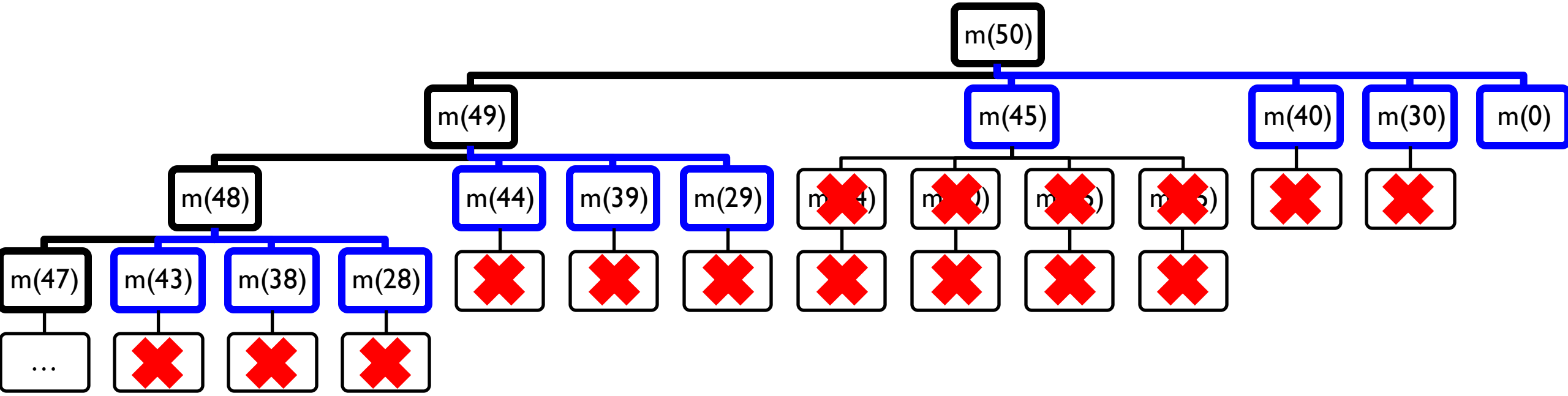
- Shortcut earlier

$$\text{Running time} = \Theta(\text{number of unique subproblems})$$

- P unique subproblems, one for every amount up to P
- $\Theta(P)$
- But this is wrong!
  - No dependence on D
  - Just as fast when $D = 1$ vs. when $D = 10^6$: is that reasonable?

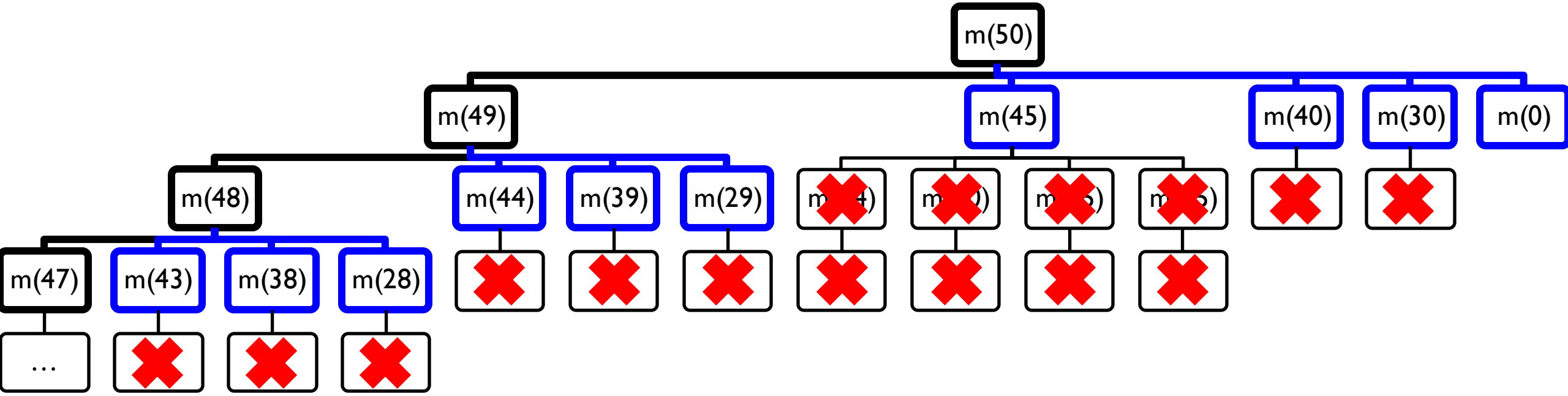
Look at the Recursion Tree Again

- Number of unique subproblems = black nodes
- But blue nodes contribute to the running time too, we also need to count them



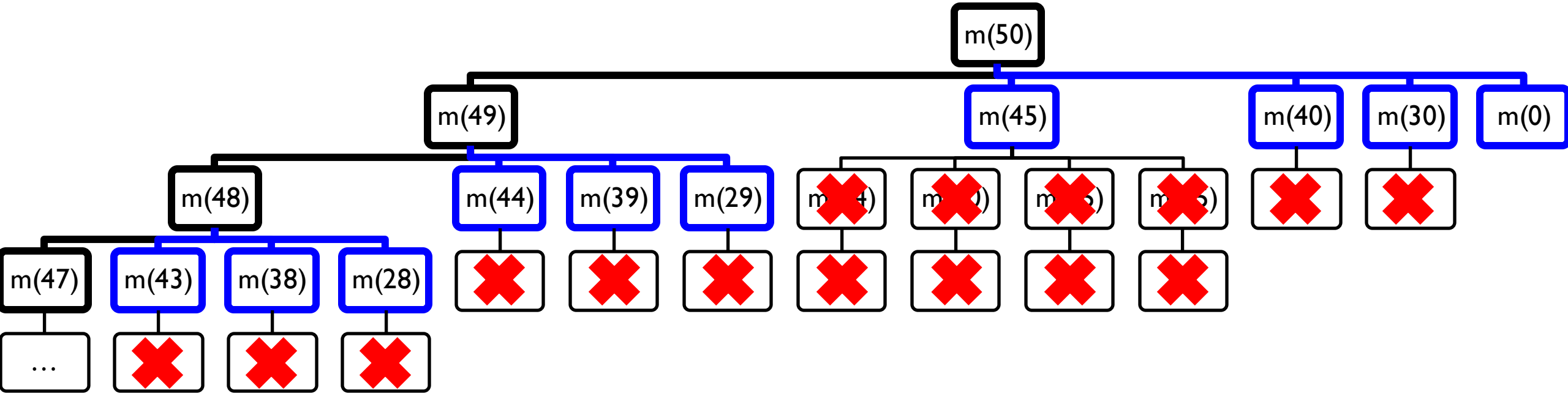
Look at the Recursion Tree Again

- Earlier, it was ok to ignore them, because the was only a small constant number of them below each black node



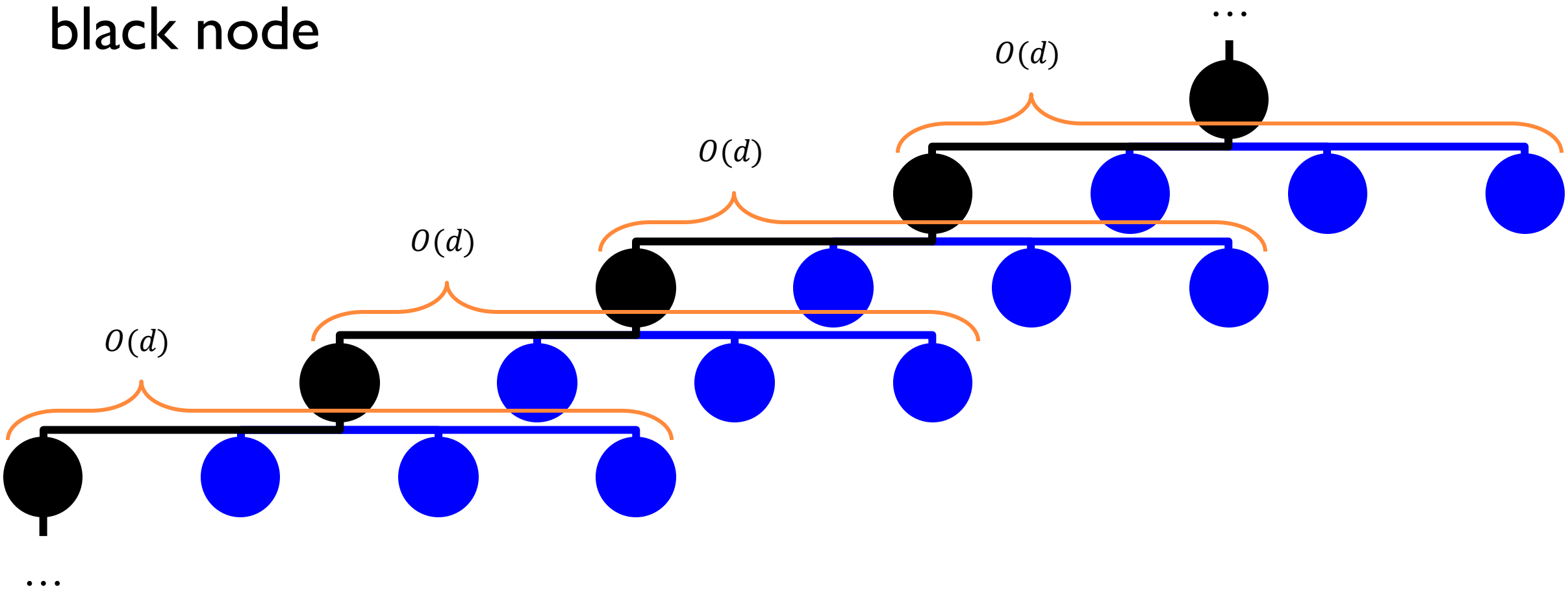
Look at the Recursion Tree Again

- P black nodes, at most D blue nodes below each black node $\rightarrow O(PD)$ blue nodes $\rightarrow O(P + PD)$ total nodes
- $O(PD)$ running time



We Can Refine Our Earlier Shortcut

- Count black nodes and blue nodes directly under each black node



We Can Refine Our Earlier Shortcut

- Running time = Θ(number of unique subproblems $\times$ number of branches per subproblem)
- Actually, slightly wrong
  - Code loops through all d denominations, not all of them producing branches because of the `if d <= amount` check
    - So, each black node contributes $\Theta(D)$ work anyway
  - Better shortcut: Running time = number of unique subproblems $\times$ non-recursive work per subproblem, assuming recursive calls are $\Theta(1)$
  - In many cases, slightly wrong shortcut above is correct or good enough and easier to think about

Producing the Actual List of Coins or Bills, Not Just Its Length

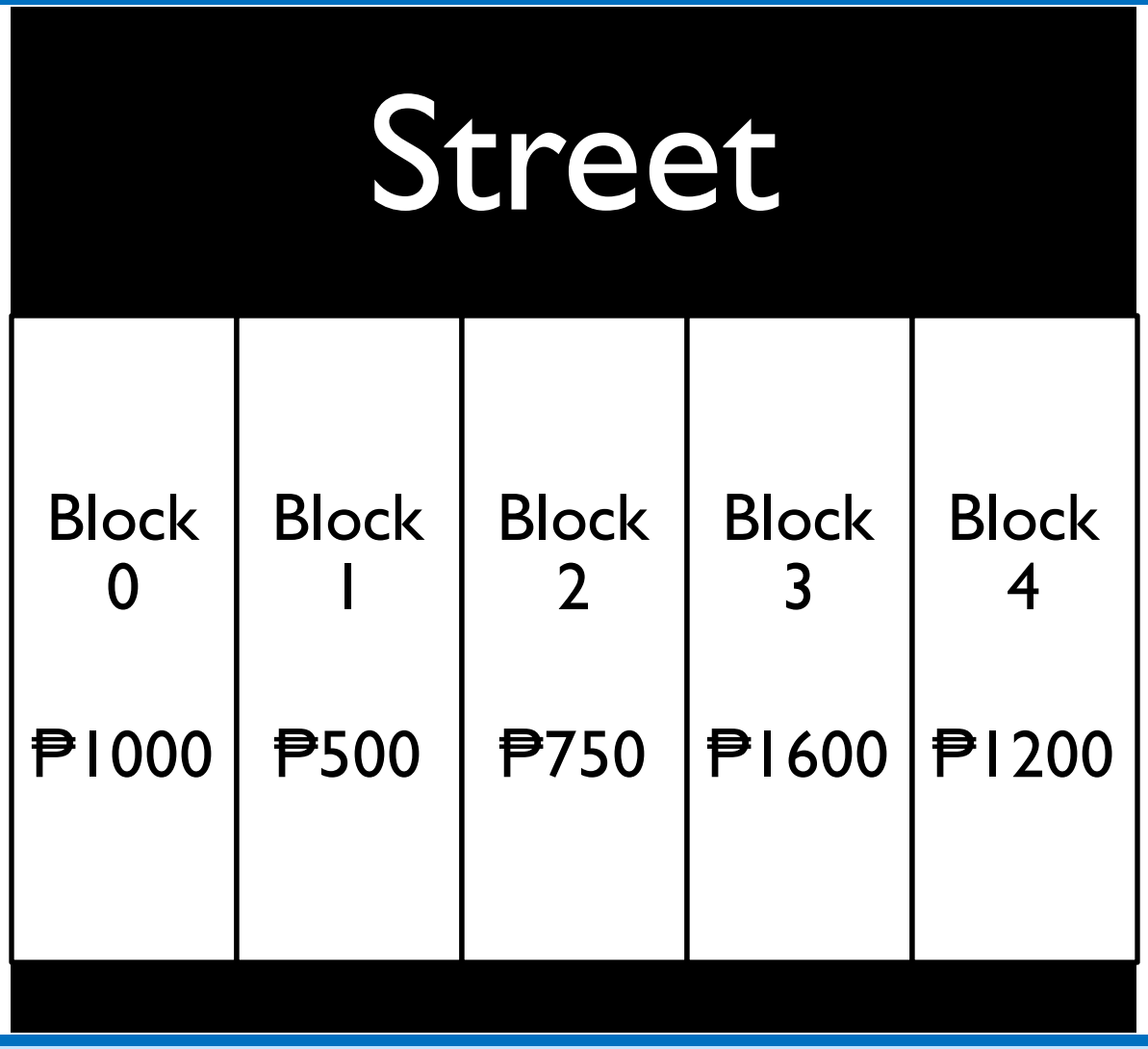
- Repeatedly pick the denomination d which results in an amount $p - d$ "closest" to 0
  - But not based on the absolute value of $p - d$
  - Rather, based on its "distance" in number of coins

```
pieces = []
while amount > 0:
  best_d = None
  for d in denominations:
    if best_d is None or min_to_make(amount - d) < min_to_make(amount - best_d):
      best_d = d
  pieces.append(best_d)
  amount -= best_d
```

Planning Store Locations

- Open convenience stores along one side of a street
- Street is divided into blocks
- For each block, we know the estimated daily profit if we open a store there
- Can't open on blocks next to each other – the two stores will compete against each other and bring profits down
- What stores to open to maximize the profit?

Example



- Open on blocks 0, 2, 4: profit ₱2950
- The following *greedy* approach does not work: open a store in the block with the highest profit and continue from there
  - Opening at block 3 yields ₱1600
  - But disallows opening at 2 or 4
  - Only remaining options are 0 or 1
  - Only one of them can be picked
  - Neither beats ₱2950

For Each Algorithm Below, Find a Counterexample

- Open stores on all even-numbered blocks
- Open stores on all odd-numbered blocks
- Open stores on either all even-numbered blocks or all odd-numbered blocks, depending on which strategy gives a higher profit

Don't Be So Smart. Just Brute Force!

- Try all valid ways to open stores
- How exactly?
  - Break down "choose some stores to open" into single decisions
    - Where to open the leftmost store
    - Then, where to open the next store to the right
    - Then, where to open the next store to the right
    - Etc.
  - A single decision is the "one step away" that reveals the recursive structure

Where to Open the Leftmost Store?

- Don't know which one is the best, just try all

```
max profit = max(
     max profit if the leftmost store is on block 0,
     max profit if the leftmost store is on block 1,
     …,
     max profit if the leftmost store is on block n - 1
)
```

How to Build the Smaller Subproblem?

- If the leftmost store is opened at block i, then the next store can't be at blocks $0,1,2, \ldots, i$
  - Because we said that the leftmost is i
- It also can't be at block $i + 1$
  - Because of the problem restriction that stores cannot be next to each other
- What is the smaller subproblem after the first step?
  - Original problem: try all ways to open stores at blocks 0 to n
  - Subproblem: try all ways to open stores at blocks $i + 2$ to n

How to Build the Smaller Subproblem?

```
max profit given list of profits P = max(
    P[0] + max profit of P without 0 to 1,
    P[1] + max profit of P without 0 to 2,
    P[2] + max profit of P without 0 to 3,
    …,
    P[n - 1] + max profit of P without 0 to n
)
```

Suggests the Following Code

```
def max_profit(P):
  return max(
    P[choice] + max_profit(P[choice+2:])
    for choice in range(len(P))
  )
```

And the Base Case

```
def max_profit(P):
  if len(P) == 0:
    return 0
  else:
    return max(
      P[choice] + max_profit(P[choice+2:])
      for choice in range(len(P))
    )
```

Making it More Efficient

- As you might guess, this gets the correct answer, but painfully slowly
- Think about it: will this have overlapping subproblems?
- This suggests we use memoization
- One problem: the parameter to the function is a list
  - How to code the memo?
    - There are ways, but they're rather ugly
  - Copying and slicing lists is slow

Instead, "Fake" the List Slicing

- Important information to keep track of: what part of the list is the subproblem looking at?
- Start and end indices
- Notice though: end index is always n
  - Subproblems are *suffixes* of the original problem

Instead, “Fake” the List Slicing

```
def max_profit(P):
  return max(
    P[choice] + max_profit(P[choice+2:])
    for choice in range(len(P))
  )

def max_profit(start):
  return max(
    P[choice] + max_profit(choice + 2)
    for choice in range(start, len(P))
  )
```

And the Base Case

```
def max_profit(start):
    if start < len(P):
        return max(
            P[choice] + max_profit(choice + 2)
            for choice in range(start, len(P))
        )
    else:
        return 0
```

- Try memozing it yourself!

Final Solution

```
memo = [None for i in range(len(P) + 2)]
def max_profit(start):
    if memo[start] is None:
        if start < len(P):
            answer = max(
                P[choice] + max_profit(choice + 2)
                for choice in range(start, len(P))
            )
        else:
            answer = 0
        memo[start] = answer
    return memo[start]
```

Things to Think About

- Why does the memo have length `len(P) + 2`?
  - `max_profit` can be called with `start = len(P) + 1`
    - When `choice` goes to `len(P) - 1`
- What is the initial call to print the answer for the original problem?
  - `print(max_profit(0))`

How Fast is This?

- How many unique subproblems?
  - $\Theta(n)$ different possible values of `start`
    - Another way to say it $\Theta(n)$ different possible suffixes
- How many branches or how much non-recursive work per subproblem?
  - $O(n)$, because of the i loop
- Therefore, how much time in total?
  - $O(n^2)$ by our shortcut

How Fast is This?

- More precisely, branches / non-recursive work is:
  - $\approx n$ when subproblem size is n
  - $\approx n - 1$ when subproblem size is $n - 1$
  - $\approx n - 2$ when subproblem size is $n - 2$
  - etc.
- Total is $1 + 2 + \cdots + n$ but this is also just $\Theta(n^2)$

An Even Faster Way

- "Choose some stores to open" can be broken down into this way instead:
  - Open store 1 or not?
  - Open store 2 or not?
  - …
  - Open store n or not?
- Smells like Lights Out
- Recursive structure: decide to open a store at `start` or not, then recursively decide the rest

An Even Faster Way

- Opening at `start` means we cannot open at `start + 1`
- To capture this constraint, exclude `start + 1` from the list of possible places for the next store
  - In other words, start index for the subproblem is `start + 2`

An Even Faster Way

```
def max_profit(start):
  if start < len(P):
    return max(
      # leftmost allowed block is chosen
      P[start] + max_profit(start + 2),

      # leftmost allowed block is not chosen
      max_profit(start + 1)
    )
  else:
    return 0
```

Convince yourself that this works.

Then, you can memoize it yourself.

An Even Faster Way

- How many unique subproblems?
  - $\Theta(n)$ different possible values of `start`
- How many branches or how much non-recursive work per subproblem?
  - Now only $\Theta(1)$
- Therefore, how much time in total?
  - Just $\Theta(n)$

Another Way to Look at What We Did

- Recursion is just a fancier loop

```
i = 0
while i < n:
    print(i)
    i = i + 1
```

```
def loop(i):
    if i < n:
        print(i)
        loop(i = i + 1)
loop(i = 0)
```

Another Way to Look at What We Did

- A "greedy" algorithm uses some smart rule to choose what the next move should be
  - The algorithm can be expressed recursively and if we do, the resulting recursion tree always has one branch per node
- Recursive backtracking / dynamic programming is just the "multiple" branches version
  - Instead of having some magic rule to decide what the next move is, try all possible next moves "in parallel"

Another Way to Look at What We Did

Making Change by Recursive Backtracking (try all branches)

```
def min_to_make(amount):
    return min(
        1 + min_to_make(amount - d)
        for d in denominations if d <= amount # many branches
    )
```

Making Change by Cashier's Algorithm (always choose one branch)

```
def min_to_make(amount):
    d = max(d for d in denominations if d <= amount)
    return 1 + min_to_make(amount - d) # only one branch
```

Another Way to Look at What We Did

Choosing Store Locations by Recursive Backtracking (try both branches)

```
def max_profit(choice):
    if choice < len(P):
        return max(P[choice] + max_profit(choice + 2), max_profit(choice + 1))
    else:
        return 0
```

Choosing Store Locations by "open on even-numbered blocks" (always choose one branch)

```
output = 0
while choice < len(P):
    output += P[choice]
    choice = choice + 2
```

```
def max_profit(choice):
    if choice < len(P):
        return P[choice] + max_profit(choice + 2)
    else:
        return 0
```

Hint for Coming Up with DP Solutions

- Usually, one of the parameters is a "loop counter" that tells us "where we are" in the input list as we construct the solution
- This parameter is used to distinguish the base case from the recursive case
- And this parameter is what advances towards the base case, just like the loop counter in a loop which advances towards the terminating condition

Printing the Actual Blocks Where to Open

- Idea: like before, repeatedly:
  - Find subproblem with the best "distance" to the base case
  - Change parameters into that subproblem

```
stores = []
start = 0
while start < len(P):
    best = None
    for choice in range(start, len(P)):
        if best is None or max_profit(choice) > max_profit(best):
            best = choice
    stores.append(best)
    start = best + 2
```

But It's Wrong!

- If you run this code, it will just print out the even-numbered blocks, which we already know is wrong
  - `max_profit(i)` is always $\geq$ `max_profit(j)` for any $i > j$
    - Remember, `max_profit(i)` means "can open at blocks i onwards"
    - The valid ways to open stores from block j onwards are also valid ways to open stores from block i onwards
    - `max_profit(i)` is considering the same possibilities as `max_profit(j)`, and others that are not valid for j
  - So, code above will repeatedly pick the leftmost allowed block to open a store

Fixing It

```
max profit given list of profits P = max(
    P[0] + max profit of P without 0 to 1,
    P[1] + max profit of P without 0 to 2,
    …,
    P[n - 1] + max profit of P without 0 to n
)
```

- The best choice depends not only on the recursive call, but also on the profit of the block being considered

Fixing It

- Let `profit_if_start(choice)` be the maximum profit gained from blocks i onwards and committing to the choice of opening a store at block i

```
def profit_if_start(choice):
    return P[choice] + max_profit(choice + 2)
```

Fixing It

- Then the correct way to construct the optimal plan is:

```
stores = []
start = 0
while start < len(P):
    best = None
    for choice in range(start, len(P)):
        if best is None or profit_if_start(choice) > profit_if_start(best):
            best = choice
    stores.append(best)
    start = best + 2
```

Why The Wrong Solution Worked Earlier

```
def min_to_make(amount):
    return min(
        1 + min_to_make(amount - 1),
        1 + min_to_make(amount - 2),
        1 + min_to_make(amount - 5)
    )
```

- The "commitment" contribution of a choice to the answer is the same for all the choices

You Are A Burglar

Item	Value	Weight
TV	₱15,000	35 kg
Computer	₱20,000	10 kg
Sack of rice	₱100	25 kg
Motorcycle	₱30,000	40 kg

Limit: 45kg

- You can carry at most 45kg in your huge knapsack
  - "Knapsack" is what older people who invented this problem called what we call "backpack"
- A house has some items, each with a value and weight
- Which items should you steal to get the most value?
  - TV + computer = ₱35,000

You Are A Professional Burglar

- Write a program that solves the problem in general, so that you can know what items to steal every night at different houses
- Of course, the program must be efficient; otherwise, you'll get caught before it finishes running!

Disclaimer

- For educational purposes only
- Don't try in real life
- A non-evil equivalent scenario:
  - Many activities to do, but limited time
  - Each activity requires some time to complete
  - Each activity gives you some amount of happiness
  - Choose which activities to do to maximize your happiness

Think

- How would you solve this?
- Any rules that come to mind on how to choose the items?
- Possible ideas:
  - Keep choosing the most valuable item first
  - Keep choosing the lightest item first

Don't Be A Greedy Burglar

Item	Value	Weight
TV	₱15,000	35 kg
Computer	₱20,000	10 kg
Sack of rice	₱100	25 kg
Motorcycle	₱30,000	40 kg

Limit: 45kg

- Best answer:
  - TV + computer = ₱35,000
- Most valuable item first:
  - Motorcycle only = ₱30,000
- Lightest item first:
  - Computer + sack of rice = ₱20,100

Dynamic Programming to the Rescue!

- Identify the "single step" in the process
- This single step is what relates a problem to a subproblem
- When there are multiple choices for a single step, try them all "in parallel"

Be A Dynamic Burglar Instead

- Items in the problem have no inherent order
- We can invent an order so that we can speak of a 0^{th} item, 1^{st} item, 2^{nd} item, etc.
- Given any list of items where the ith item has weight $W[i]$ and value $V[i]$, choose items so that the total value is maximized and the total weight does not exceed a limit L
- This allows us to have a "loop counter" or *measure of progress* in the recursion

Be A Dynamic Burglar Instead

- Breaking down the process into steps:
  - Pick first item to steal
  - Pick second item to steal (*must be to the right of the first*)
  - Pick third item to steal (*must be to the right of the second*)
  - Etc.
- This looks like store planning!

Be A Dynamic Burglar Instead

```
max value = max(
    max value when first chosen item is item 0,
    max value when first chosen item is item 1,
    …,
    max value when first chosen item is item n - 1
)
```

Be A Dynamic Burglar Instead

```
def best_value(start):
  if start < len(V):
    return max(
      V[choice] + best_value(choice + 1)
      for choice in range(start, len(V))
    )
  else:
    return 0
```

Something's Missing

- If you run this code, it always just returns the sum of the values of all the items
- We forgot to include the weight limit in the code

An Idea

- Keep track of what items have been selected so far
- So that we can check if a combination is valid at the end

An Idea

```
def best_value(start, chosen):
  global limit
  if start < len(V):
    return max(
      V[choice] + best_value(choice + 1, chosen + [choice])
      for choice in range(start, len(V))
    )
  else:
    return 0 if sum(W[j] for j in chosen) <= limit else ???
```

What to Return When Some Choice is Invalid?

- $-\infty$, so that the invalid branch gets eliminated by the `max` function
- Think about it: why don't we just return, for example, -1?

What to Return When Some Choice is Invalid?

```
def best_value(start, chosen):
  global limit
  if start < len(V):
    return max(
      V[choice] + best_value(choice + 1, chosen + [choice])
      for choice in range(start, len(V))
    )
  else:
    return 0 if sum(W[j] for j in chosen) <= limit else -math.inf
```

There's an Issue

- There's a parameter in the function, `chosen`, which is a list
- Memoizing this is difficult
- Even more serious issue: `chosen` ensures no overlapping subproblems, making memoization useless!

DP vs. Backtracking

- DP is "careful" brute force
- Unlike backtracking where we can use as many variables as we want, in DP, we should try to "remember" as little information as possible in the function parameters
  - While still having a way to eliminate invalid solutions
- More parameters ⇒ more subproblems ⇒ lower chances of overlapping ⇒ slower (even when memoized)

Remembering the Choices is Too Much Info

- Only the *sum* of the weights of the choices is needed to check if a solution is valid or not

Only Remember the Sum

```
def best_value(start, total_weight):
  global limit
  if start < len(V):
    return max(
      V[choice] + best_value(
        choice + 1,
        total_weight + W[choice]
      )

      for choice in range(start, len(V))
    )
  else:
    return 0 if total_weight <= limit else -math.inf
```

There's Another Way to Write This, Can You Explain Why It's Bad?

```
def best_value(start, total_weight, total_value):
  global limit
  if start < len(V):
    return max(
      best_value(
        choice + 1,
        total_weight + W[choice],
        total_value + V[choice]
      )
      for choice in range(start, len(V))
    )
  else:
    return total_value if total_weight <= limit else - math.inf
```

Steps of the Solution to Try Yourself

- Memoize!
- Constructing the actual list of items once we have a DP solution for getting the maximum value
  - Notice: usually, once you have a DP solution that gets the best "score" of a solution, getting the actual solution is much easier
  - Good idea to focus on solving for the "score" first and then worry about constructing the actual solution later, or not at all

Running Time Analysis (Assuming Memoized)

- Only thing changed from store planning problem: extra parameter
- Changes the number of subproblems from $\Theta(n)$ to $\Theta(nS)$
  - Where S is the maximum possible total weight of any one branch: in other words, the total weight of all items
- Work per subproblem: $\Theta(n)$
- Total: $\Theta(nS) \times O(n) = O(n^2S)$

A Slight Improvement

- We can do slightly better by **pruning** branches that already go over the limit

A Slight Improvement

```
def best_value(start, total_weight):
  global limit
  if start < len(V):
    return max(
      V[choice] + best_value(choice + 1, total_weight + W[choice])
      for choice in range(start, len(V))
      if total_weight + W[choice] <= limit
    )
  else:
    return 0
```

Something to Think About

- Why does the base case just return 0, no more check?

Running Time Analysis (Assuming Memoized)

- Work per subproblem is still the same: $\Theta(n)$
- But now, the number of subproblems is $\Theta(nL)$ instead of $\Theta(nS)$, where L is the carrying limit
  - L can be much smaller than S, the total weight of all the items
  - More technically, it is $\Theta(n \times \min(L, S))$
- So, the total work is $\Theta(n^2 L)$
  - Or $\Theta(n^2 \times \min(L, S))$ if you want to be more technical

Try It Yourself

- Earlier, we found a way to improve the solution for store planning from $\Theta(n^2)$ to $\Theta(n)$
- Same idea can be used to improve the solution for this **knapsack problem** from $\Theta(n^2 L)$ to $\Theta(nL)$

Practice Problems

- https://progvar.fun/problemsets/dp-classic

Thanks!


accenture
PLDT
stratpoint
FAST FORWARD TO THE FUTURE
MERALCO
Thinking Machines
Data Science