Dynamic Programming

An Introduction to DP

Dynamic Programming?

- A programming technique
  - Solve a problem by breaking into smaller sub-problems
  - Similar to recursion with memoisation
- Usefulness: Efficiency
  - Exponential to Polynomial
  - Trades memory for speed
- Frequently used in Olympiads

Fibonacci Numbers

- A sequence where every number is the sum of the previous 2
- 1, 1, 2, 3, 5, 8, 13, …
- What is the N^{th} Fibonacci number, F(N)?
  - We will solve this using several different techniques

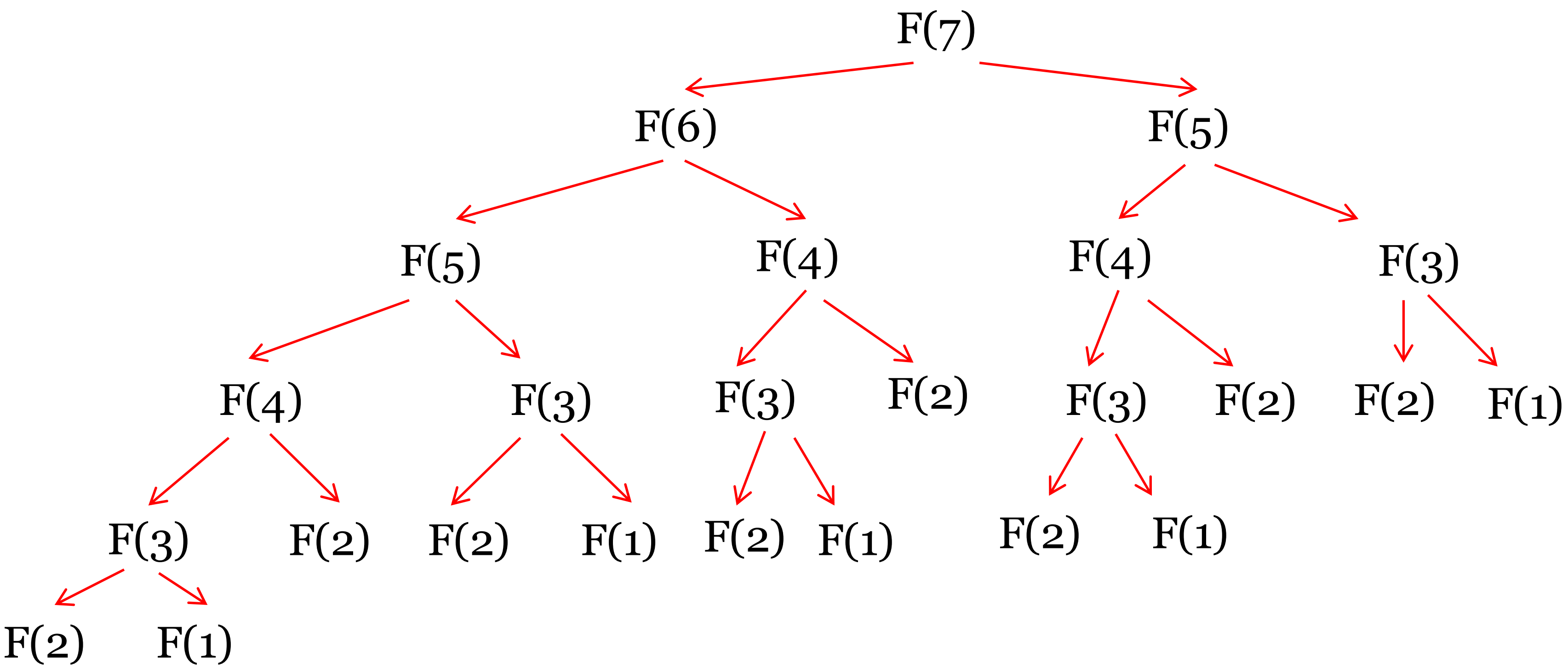
Fibonacci Numbers: Recursion

- Split problem into smaller sub-problems
  - F(N) = F(N-1) + F(N-2)
- Solve the smaller sub-problems:
  - F(N-1) = F(N-2) + F(N-3)
  - etc.
- Terminates when we reach the base case
  - F(1), F(2) are defined to be 1

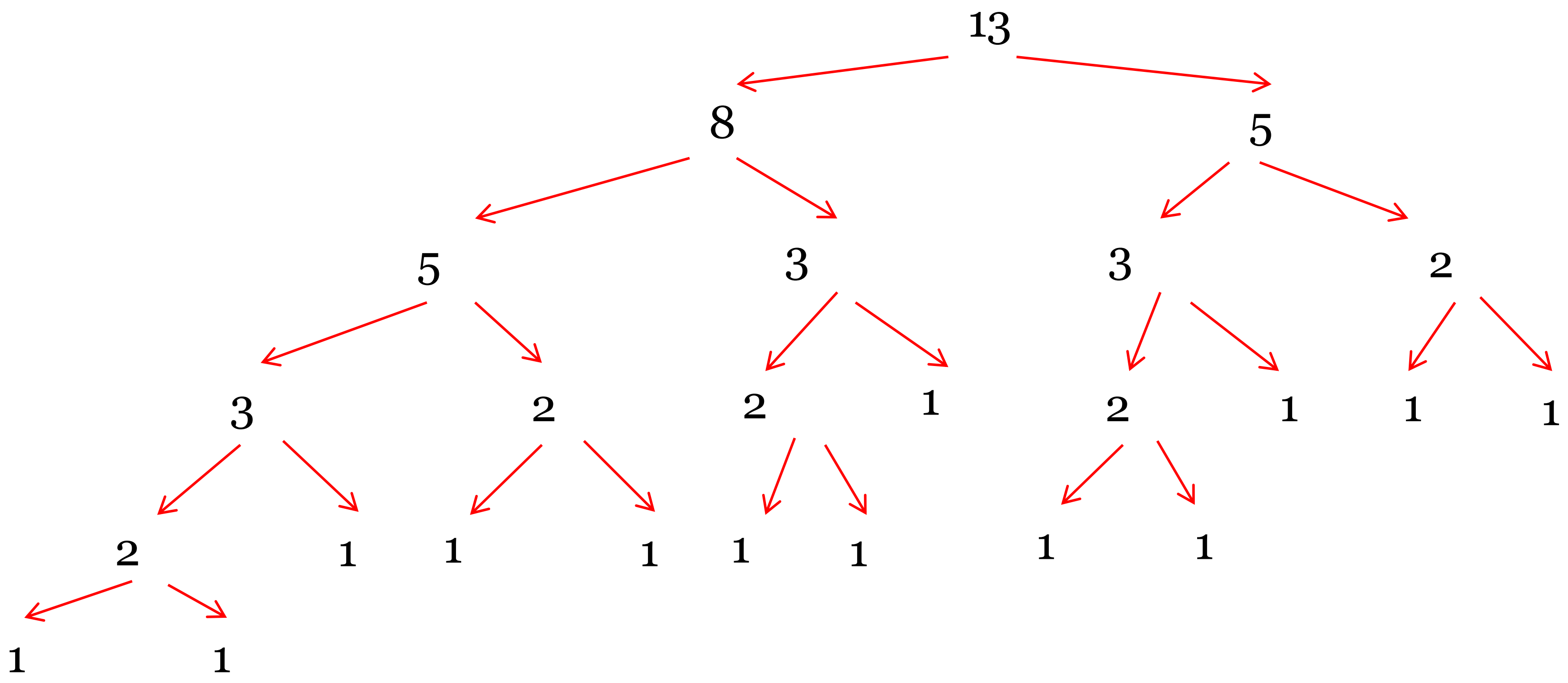
Fibonacci Numbers: Recursion

```
int fibonacci(int n)
{
        if (n <= 2)
        return 1;
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

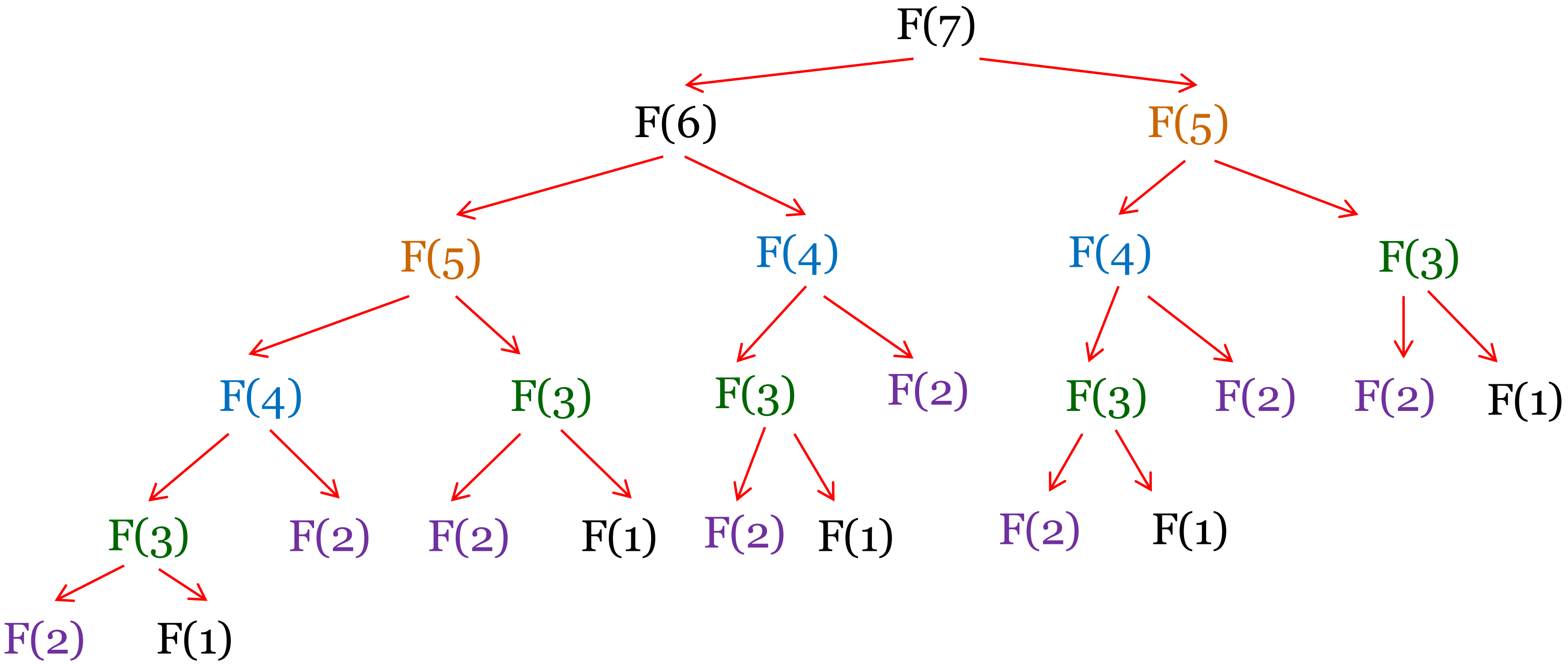
Fibonacci Numbers: Recursion



Fibonacci Numbers: Recursion



Fibonacci Numbers: Recursion



Many repeated recursive calls!

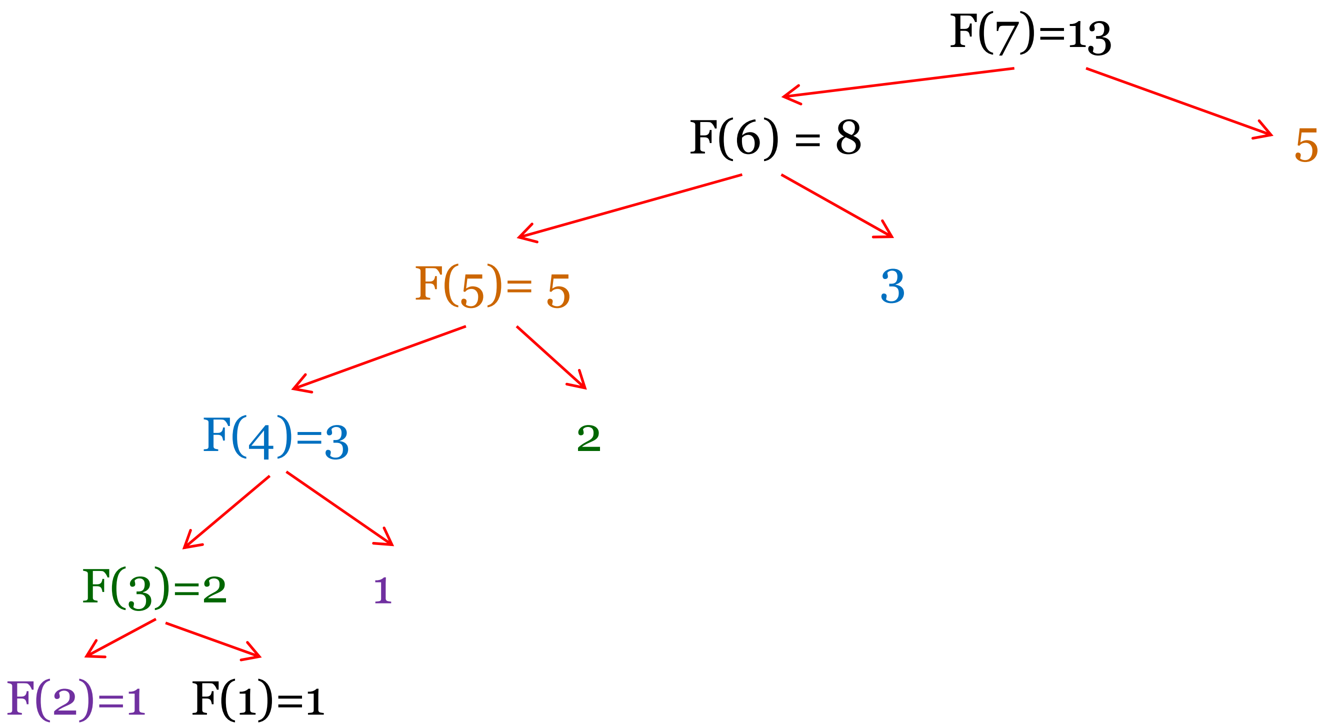
Fibonacci Numbers: Recursion

- Exponential time complexity – bad!
- The cause: repeated sub-problems
- Solution: store the results of each sub-problem
  - Trade memory for speed

Fibonacci Numbers: Memoisation

- Optimisation technique that avoids repeated function calls
  - When we find F(x), store it
  - Next time we need it, use stored result

Fibonacci Numbers: Memoisation

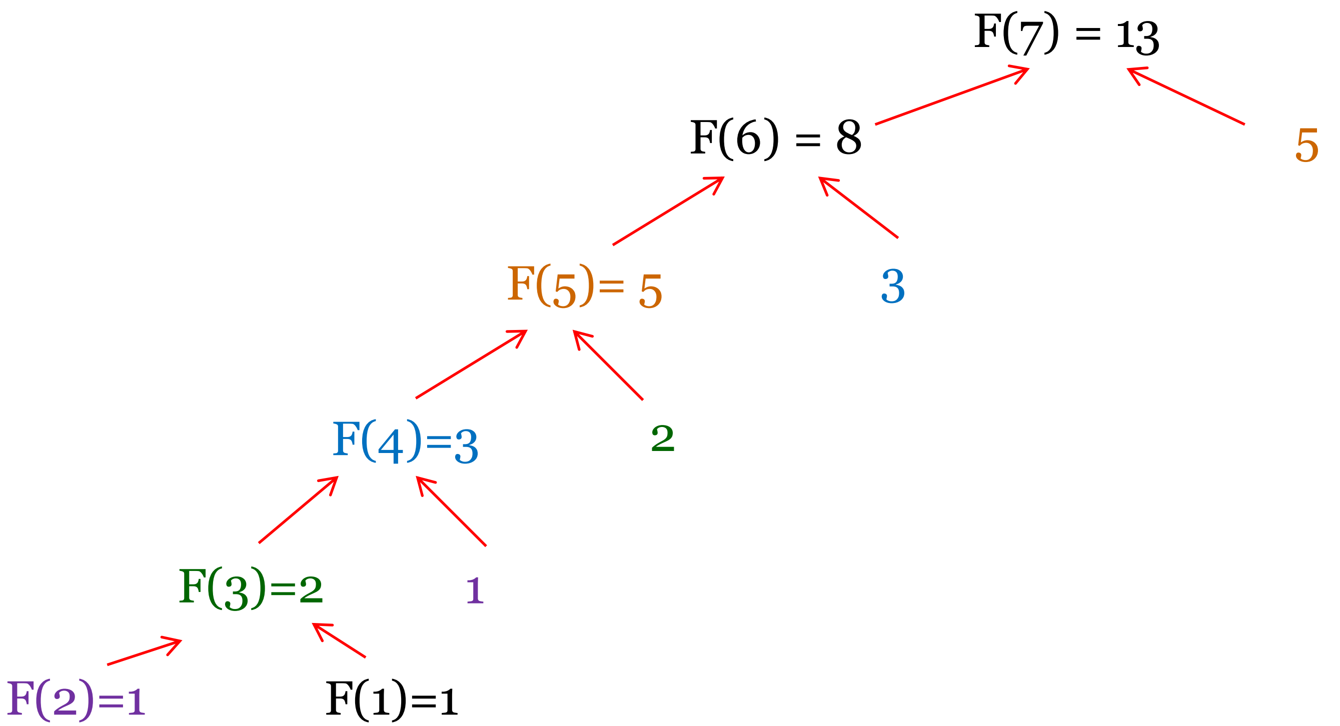


Exponential to Linear!

Fibonacci Numbers: DP

- Memoisation, but bottom-up
  - Start from base case
  - Build up to the given problem

Fibonacci Numbers: DP



Efficiency class: O(N)

Fibonacci Numbers: DP

```
int fib(int n)
{
    int f[n+1];
    f[0] = 1;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
            f[i] = f[i – 2] + f[i – 1];
    return f[n];
}
```

Fibonacci Numbers

- Our techniques require breaking the problem into smaller sub-problems
  - Used the relation F(N) = F(N-1) + F(N-2)
  - Always reaches base case
- The output F(N) only depends on the input N
  - So bottom-up works
- DP faster

How to DP

- Identify the recurrence relation/dependency
- Construct a recursive function as the solution
  - The answer must depend only on the parameters
  - A 'mathematical' function, e.g. F(N)
  - Use as few parameters as possible
- Use an array to store the results
  - Multi-dimensional? (One for each parameter)
- Nested Loops from base case to given problem
  - Order must satisfy dependencies

DP vs Recursion

- Advantages:
  - Speed
  - Code simpler
- Disadvantages:
  - Memory (multi-dimensional!)
  - Conceptually more difficult
  - Not always possible

DP vs Recursion with Memoisation

- Theoretically equivalent
- Same time complexity
- Bottom-up vs Top-down
- Advantages:
    - Less memory
        - Stack + function call overhead
        - Memory saving trick (later)
- Disadvantages:
    - Conceptually more difficult
        - Complicated dependencies?

Another example: Coin Counting

- We want to make M cents of change
- N different types of coins are available (V[1]...V[N])
- Least number of coins?

Coin Counting

- Dependency:
  - coins(M) = 1+ min {coins(M-V[1]),…,coins(M-V[N])}
  - Invalid coins(M): no smaller problems solved
  - Base case: coins(0) = 0
- Implementation
  - A coins array with coins[0] = 0
  - Everything else initialised to -1
  - Loop from 1 to M, using the dependency for coins[i]

Coin Counting

M	0	1	2	3	4	5	6	7
Min # coins	0	-1	1	1	2	1	2	2

Given coins (V[N]): {2,3,5}

Coin Counting

```
int N, M;
int V[N];
int coins[M + 1];

set(coins[0], coins[M], -1);
coins[0] = 0;
for (int i = 1; i <= M; i++)
{
    int best = M;
    for (int j = 0; j < N; j++)
            if (V[j] <= i && coins[i – V[j]] != -1 && coins[i – V[j]] + 1 < best)
                    best = coins[i – V[j]] + 1;
    coins[i] = best;
}
```

Backtracking

- Unnecessary info suggests DP
- But sometimes, require the ‘path’ to the solution
- Coin Counting:
  - Find the minimum number of coins
  - But also output which coins they are

Backtracking

- General: For each value from base to M:
  - Use array as before
  - But also use an array to store path
    - Memory concerns
- Coins: For each value from 0 to M:
  - Store min # coins
  - Store last coin used
    - Can *backtrack* to find path from 0 to M
    - Trade speed for memory

Backtracking: Coin Counting

M	0	1	2	3	4	5	6	7
Min # coins	0	-1	1	1	2	1	2	2
Last coin	-1	-1	2	3	2	5	3	2

Given coins (V[N]): {2,3,5}

Backtracking: Coin Counting

M	0	1	2	3	4	5	6	7
Min # coins	0	-1	1	1	2	1	2	2
Last coin	-1	-1	2	3	2	5	3	2

Given coins (V[N]): {2,3,5}

'Path': {5,2}

Backtracking: Coin Counting

```
int N, M;
int V[N];
int coins[M + 1];
int coinUsed[M + 1];

coins[0] = 0;
for (int i = 1; i <= M; i++)
{
    int best = M;
    int coin = -1;
    for (int j = 0; j < N; j++)
            if (V[j] <= i && coins[i – V[j]] + 1 < best)
            {
                    best = coins[i – V[j]] + 1;
                    coin = j;
            }
    coins[i] = best;
    coinUsed[i] = coin;
}
```

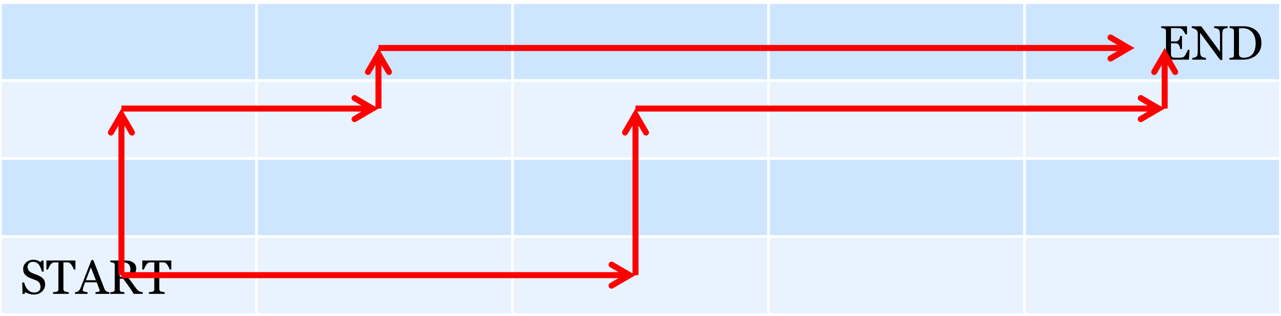
Less memory, more time…

Multi-Dimensional DPs

- So far, 1D
  - F[N] = F[N-1] + F[N-2]
  - Coins[M]=1+ min {coins(M-V[1]),...,coins(M-V[N])}
- 2D or more often required

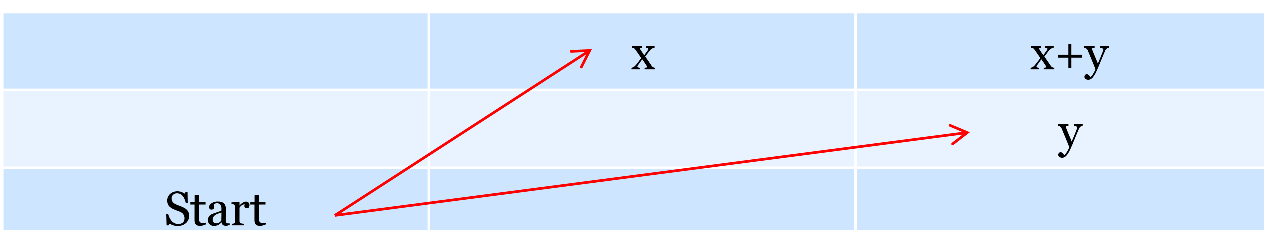
Example: Number of paths

You start at the bottom left of a NxM rectangular grid, and can only move upward or right. How many ways are there of getting to the top right corner?



Number of Paths

- Want the # paths from start to end
- State for DP: # paths from start to any given square
- Identify the dependency
  - Can only get to a square from below or the left
  - There is no overlap from below or from left
  - # ways to get to a square is the sum

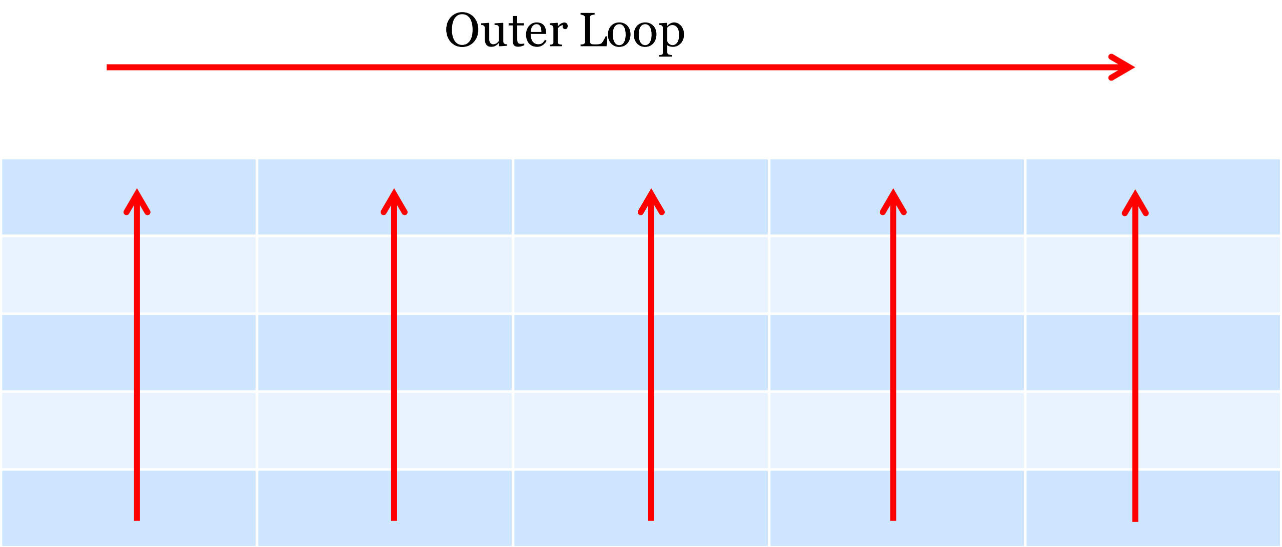


Number of Paths

- Dependency:
  - paths[width][height] = paths[width-1][height] + paths[width][height-1]
  - 2D recurrence relationship
- Having identified this:
  - Construct the recursive function
  - Use a 2D array to store results
  - Use nested looping in a valid order to populate array

Number of Paths

- Use nested looping in a valid order



Number of Paths

- Use nested looping in a valid order

Outer Loop →

1	5	15	35	70
1	4	10	20	35
1	3	6	10	15
1	2	3	4	5
1	1	1	1	1

Memory Saving Technique

- Array for all values is inefficient
  - May be too large
  - Particularly for > 1D
- Store only subset of the parameter space
- Dependency determines which values needed
- Like a slider
  - Change the letter if 3/5 chars before are ‘T’:
    - T F T T F T F F T F T T T F T F

Memory Saving Technique

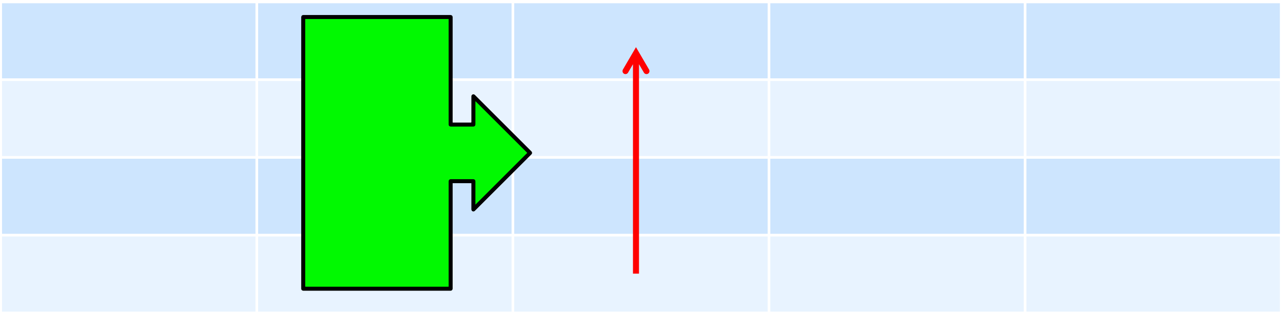
- Fibonacci:
  - F(N) = F(N-1) + F(N-2)
- Only need previous 2 values
  - Array unnecessary

Memory Saving Technique

```
int fib (int n)
{
    int f1, f2 = 1;
    for (int i = 2; i <= n; i++)
    {
            int temp = f2;
            f2 = f1 + f2;
            f1 = temp;
    }
    return f2;
}
```

Memory saving technique

- More relevant for higher dimensions
- Often store only the last row, or last 2 rows, etc.
- Number of paths:
  - Only previous column needed



DP: The difficulty

- Knowing what to DP on (which dependency/ 'state'?)
  - Which parameters to use
  - Sometimes use DP for a sub-problem only
- Finding the relation/dependency

How to Identify a DP Problem

- Typical Traits:
  - Some main integer variables, e.g. N
  - Neither large nor very small (30 < N < 10000)
  - O(N^2) or O(N^3) acceptable
- 'States' exist (configurations/situations)
  - Higher states can be derived from lower states
- These are only rough rules of thumb
  - No fool-proof rules exist

Example: Subset Sums

- For many sets of consecutive integers from 1 through N (1 <= N <= 39), one can partition the set into two sets whose sums are identical.
- For example, if N=3, one can partition the set {1, 2, 3} in one way so that the sums of both subsets are identical: {3} and {1,2}
- Reversing the order counts as the same partitioning
- If N=7, there are four ways to partition the set {1, 2, 3, ... 7} so that each partition has the same sum:
  - {1,6,7} and {2,3,4,5}
  - {2,5,7} and {1,3,4,6}
  - {3,4,7} and {1,2,5,6}
  - {1,2,4,7} and {3,5,6}
- Given N, your program should print the number of ways a set containing the integers from 1 through N can be partitioned into two sets whose sums are identical. Print 0 if there are no such ways.

Reminder: How to DP

- **Identify the state & recurrence relation**
- **Construct a recursive function as the solution**
  - The answer must depend only on the parameters
  - A 'mathematical' function, e.g. F(N)
  - Use as few parameters as possible
- Use an array to store the results
  - Multi-dimensional? (One for each parameter)
- Nested Loops from base case to given problem
  - Order must satisfy dependencies

Subset Sums

- State:
  - partitions(N,D) counts the # of partitionings of {1,2,…,N} into two sets which differ by D
  - $D \leq N(N+1)/2$

Subset Sums

- State:
  - Partitions(N,D) counts the # of partitionings of {1,2,…,N} into two sets which differ by D
  - $D \leq N(N+1)/2$
- Dependency:
  - $p(N,|D|) = p(N-1,|D-N|) + p(N-1,|D+N|)$
    - If we remove the no. 'N', we need the difference between the remaining sets to be $D \pm N$
- This was the difficult part

Reminder: How to DP

- Identify the state & recurrence relation
- Construct a recursive function as the solution
  - The answer must depend only on the parameters
  - A 'mathematical' function, e.g. F(N)
  - Use as few parameters as possible
- **Use an array to store the results**
  - Multi-dimensional? (One for each parameter)
- **Nested Loops from base case to given problem**
  - Order must satisfy dependencies

Subset Sums

- Base case: N=1
  - p[1][1] = 1
  - p[1][x] = 0 for other x
- Nested looping in a valid order:
  - Need all p[N-1][i] before any p[N][j]
  - Loop from N = 0 to N = problem size
    - For each N, find p[N][D] for each D

Subset Sums

D | N	1	2	3
0	0	0	1
1	1	1	0
2	0	0	1
3	0	1	0
4	0	0	1
5	0	0	0
6 = N(N+1)/2	0	0	1

Outer loop