The Soul of Dynamic Programming
Formulations and Implementations

Bowen Yu

March 1, 16
Special Topic for Algorithmic Problem Solving
What is DP all about

• Advanced technique? 😊

• Transition Function
• Memoization
An example that everyone uses

- Calculate the Fibonacci numbers
- \( F(i) = F(i-1) + F(i-2) \)

```cpp
def fib(int i):
    if i <= 1:
        return i;
    return fib(i - 1) + fib(i - 2);
```
We know this is slow
Optimization?

- No need to compute a same value more than once

```java
int memo[MAXN];
int fib(int i) {
    if (memo[i] != -1) return memo[i];
    if (i <= 1) return memo[i] = i;
    return memo[i] = fib(i - 1) + fib(i - 2);
}
```
But...

- Why the -1's?

```c
30 int fib[MAXN];
31 void allFibs() {
32     fib[0] = 0;
33     fib[1] = 1;
34     for (int i = 2; i < MAXN; i++) {
35         fib[i] = fib[i - 1] + fib[i - 2];
36     }
37 }
```
So-called top-down and bottom-up

• Top-down:
  – Derive the DP entry dependencies
  – Memoize and compute recursively

```cpp
int memo[MAXN];
int fib(int i) {
    if (memo[i] != -1) return memo[i];
    if (i <= 1) return memo[i] = i;
    return memo[i] = fib(i - 1) + fib(i - 2);
}
```

• Bottom-up:
  – Compute values in specific order
  – No memoization involved

```cpp
int fib[MAXN];
void allFibs() {
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i < MAXN; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
}
```
Two different DPs?

• Not really.

• They are actually the same computation, as they have exactly the same transitions $F(i) = F(i-1) + F(i-2)$.

• We need to get to the soul of DP to actually see them unified.
Revisiting DP

- **State** – an entry of some value of your interest
- **Transition** – relations between the states
- **Order** – how states get computed
Fibonacci Sequence

• **State**
  – Fib(i)

• **Transition**
  – Fib(i) = Fib(i – 1) + Fib(i – 2)

• **Order?**
  – We always have larger Fibs depending on smaller Fibs.
  – Therefore, we can compute smaller Fibs first before computing larger Fibs.
Thinking on a graph

• This is a **Directed Acyclic Graph (DAG)**
How DAG works?

- We can always find a **topological order** of a DAG.

- The states on the graph can be computed in this topo order, so that when a state is being computed, its dependencies would have already been computed before.

- A DP must have a corresponding DAG (most of the time implicit), otherwise we cannot find a valid order for computation.
Counting Problem

• Given a set of **N distinct positive** integers: \(v_1, v_2, \ldots v_N\)
• How many ways can we compose an **ordered list** with sum \(S\), by choosing the values from the set?
  
  (Each value can be chosen multiple times)

• For the discussion purpose, today we:
  
  – assume all integers values in this lecture are reasonably small so that they fit well in the memory, e.g. in this case \(N \leq 20\), \(v_i \leq 100\).
  
  – ignore integer overflows, which are typically handled by modulo arithmetic
Counting Problem

- **State**
  - $f(S)$: the number of ways we can compose sum $S$

- **Transition**
  - $f(s) = \sum_i f(s - v_i)$

- **Order**
  - In increasing $S$
Counting Problem – DAG (top)

\[ S \]

\[ S = S - v_1 \]
\[ S = S - v_2 \]
\[ \ldots \ldots \]
\[ S = S - v_n \]

\[ S_{11} = S_1 - v_1 \]
\[ S_{12} = S_1 - v_2 \]
\[ S_{21} = S_2 - v_1 \]
\[ S_{13} = S_1 - v_3 \]
\[ \ldots \ldots \]
Counting Problem – DAG (bottom)

\[ S_{11} = v_1 + v_1 \]
\[ S_{12} = v_1 + v_2 \]
\[ S_1 = v_1 \]
\[ S_2 = v_2 \]
\[ \ldots \]
\[ S_n = v_n \]
\[ S = 0 \]
Two implementations

- Top-down

```c
int N, v[MAXN], dp[MAXS];
int count(int S) {
    if (dp[S] != -1) return dp[S];
    int ans = 0;
    for (int i = 0; i < N; i++) {
        if (S >= v[i]) ans += count(S - v[i]);
    }
    return dp[S] = ans;
}
```

- Bottom-up

```c
int N, v[MAXN], dp[MAXS];
void count() {
    dp[0] = 1;
    for (int S = 1; S < MAXS; S++) {
        for (int i = 0; i < N; i++) {
            if (S >= v[i]) dp[S] += dp[S - v[i]];
        }
    }
}
```

Not much difference 😊
Two topo-sort algorithms

* Assuming the graph is DAG

• **Queue (Kahn’s)**
  – Add nodes with in-degree = 0 to the queue
  – Pop nodes out of the queue, decrement their neighbors’ in-degrees. Add those new nodes with in-degree = 0 to the queue.

• **DFS**
  – The current node that completes its DFS has no more outgoing edges and therefore comes first in topo order.
  – Do DP computation in post-order traversal.
Topo-sort and DP

- Use Kahn’s algorithm for topo-order: bottom-up DP

```cpp
int N, v[MAXN], dp[MAXS];
void count() {
    dp[0] = 1;
    for (int S = 1; S < MAXS; S++) {
        for (int i = 0; i < N; i++) {
            if (S >= v[i]) dp[S] += dp[S - v[i]];
        }
    }
}
```

- Use DFS for topo-order: top-down DP

```cpp
int N, v[MAXN], dp[MAXS];
int count(int S) {
    if (dp[S] != -1) return dp[S];
    int ans = 0;
    for (int i = 0; i < N; i++) {
        if (S >= v[i]) ans += count(S - v[i]);
    }
    return dp[S] = ans;
}
```
Which one is better?

• It depends:
  – If the problem has a clear implicit topo-order (e.g. the Fibonacci Sequence, the counting problem), then bottom-up could be simpler to implement.
  – If the problem has a clear sub-dividing strategy, then top-down could be more intuitive.
Cutting Sticks

• Given a stick of integer length L, cut it into multiple segments of integer lengths.
• You are given a list of (length, value) pairs. For every possible length there is an associated value.
• Your task is to maximize your total value after cutting.

• Example:

<table>
<thead>
<tr>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Answer with L = 4 is 9.
Cut into 2 segments with lengths 1 and 3.
Cutting Sticks

- **State**
  - $\text{Cut}(L)$: maximum total value we can get for a stick of length $L$

- **Transition**
  - $\text{cut}(L) = \max_i \{\text{value}(i) + \text{cut}(L - i)\}$

- **Order**
  - Larger $L$ depends on smaller $L$’s
* Note that values are **NOT** states
* Note that values are NOT states
• Because of the action of cutting, it seems that using top-down is more intuitive.

```c
30  int value[MAXN], dp[MAXN];
31  int cut(int L) {
32    if (dp[L] != -1) return dp[L];
33    if (L == 0) return dp[L] = 0;
34    int ans = 0;
35    for (int i = 1; i <= L; i++) {
36      ans = max(ans, value[i] + cut(L - i));
37    }
38    return dp[L] = ans;
39  }
```
• Actually the DAG is a complete DAG.
Cutting sticks – bottom-up implementation

- So it could be intuitive too to compute bottom up 😊

```c
31 int value[MAXN], dp[MAXN];
32 void cut() {
33     dp[0] = 0;
34     for (int L = 1; L < MAXL; L++) {
35         for (int i = 1; i <= L; i++) {
36             dp[L] = max(dp[L], value[i] + dp[L - i]);
37         }
38     }
39 }
```
Why DAG is important

- It helps you *perceive* the problem better.
- It helps you find the simplest way to *implement* the same DP algorithm.
- It helps you *determine* if a problem can be solved by DP.
Grid Walking

• Given a grid map with blocks, determine the number of different ways to walk from (top, left) to (bottom, right) if you can only walk east or south.
Perceive – Grid Walking
• **State**
  – ways(i, j): number of ways to reach cell(i, j) from the starting cell

• **Transition**
  – ways(i, j) = ways(i - 1, j) + ways(i, j - 1)
  – ways(i, j) = 0 if (i, j) is a blocked cell

• **Order**
  – Right depends on left
  – Bottom depends on top
Implement – Grid Walking

- The grid map has implicit topology that allows us to get a topo-order simply by iterating i and j.

```cpp
bool block[MAXN][MAXN];
int N, dp[MAXN][MAXN] = {};

int walk() {
    dp[1][1] = 1;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            if (block[i][j]) continue;
            dp[i][j] += dp[i - 1][j];
            dp[i][j] += dp[i][j - 1];
        }
    }
    return dp[N][N];
}
```
Implement – Grid Walking (cont’d)

• Won’t be much work to do top-down either?

```c
30  bool block[MAXN][MAXN];
31  int N, dp[MAXN][MAXN];
32  bool out(int i, int j) {
33      return i < 1 || j < 1 || i > N || j > N;
34  }
35  int walk(int i, int j) {
36      if (dp[i][j] != -1) return dp[i][j];
37      if (i == 1 && j == 1) return dp[i][j] == 1;
38      if (block[i][j] || out(i, j)) return dp[i][j] = 0;
39      return dp[i][j] = walk(i - 1, j) + walk(i + 1, j);
40  }
```
Determine – Grid Walking

• The previous Grid Walking problem clearly has a DAG.

• What if we change the possible walking directions to all four? (up, down, left, right)

• Of course we would have infinite number of ways to reach the goal. Therefore we restrict our walk to take exactly \( K \) steps.
  – Count how many ways to reach cell \((i, j)\) from \((1, 1)\) with exactly \( K \) moves
Unfortunately this is no longer a DAG and there is NO way to do DP on this.
What should we do if it is not a DAG

• Try to make a DAG if you are not given a DAG directly.

• How?
  – Observe where we cannot go back to previous states
  – In other words, observe the monotonicity based on which we can create a DAG
Grid Walking 2 – DAG creation

• Where we cannot go back
  – After taking K steps, we can never go back to the states where we have taken k < K steps.
  – In other words, K is monotonically increasing.

• Idea: Build a DAG with states ways(i, j, k): the number of ways we can reach cell (i, j) using exactly k steps.
Grid Walking 2 - DAG

Increasing $K$

38
How to implement

• Typically, one more dimension in your DP state corresponds to:
  – Top-down: a new argument of your recursive function
  – Bottom-up: a new dimension of your multi-dimensional array
bool block[MAXN][MAXN];
int N, dp[MAXN][MAXN][MAXK];

bool out(int i, int j) {
    return i < 1 || j < 1 || i > N || j > N;
}

int walk(int i, int j, int k) {
    if (dp[i][j][k] != -1) return dp[i][j][k];
    if (k == 0) return i == 1 && j == 1;
    if (block[i][j] || out(i, j)) return dp[i][j][k] = 0;
    return dp[i][j][k] =
        walk(i + 1, j, k - 1) +
        walk(i - 1, j, k - 1) +
        walk(i, j + 1, k - 1) +
        walk(i, j - 1, k - 1);
}
```c
bool block[MAXN][MAXN];
int N, dp[MAXN][MAXN][MAXK];

void walk() {
    dp[1][1][0] = 1;
    for (int k = 1; k < MAXK; k++) {
        for (int i = 1; i <= N; i++) {
            for (int j = 1; j <= N; j++) {
                if (block[i][j]) continue;
                dp[i][j][k] += dp[i + 1][j][k - 1];
                dp[i][j][k] += dp[i - 1][j][k - 1];
                dp[i][j][k] += dp[i][j + 1][k - 1];
                dp[i][j][k] += dp[i][j - 1][k - 1];
            }
        }
    }
}
```
Which one is better? 😊

```c
bool block[MAXN][MAXN];
int N, dp[MAXN][MAXN][MAXK];
bool out(int i, int j) {
    return i < 1 || j < 1 || i > N || j > N;
}

int walk(int i, int j, int k) {
    if (dp[i][j][k] != -1) return dp[i][j][k];
    if (k == 0) return i == 1 && j == 1;
    if (block[i][j] || out(i, j)) return dp[i][j][k] = 0;
    return dp[i][j][k] =
        walk(i + 1, j, k - 1) +
        walk(i - 1, j, k - 1) +
        walk(i, j + 1, k - 1) +
        walk(i, j - 1, k - 1);
}

void walk() {
    dp[1][1][0] = 1;
    for (int k = 1; k < MAXK; k++) {
        for (int i = 1; i <= N; i++) {
            for (int j = 1; j <= N; j++) {
                if (block[i][j]) continue;
                dp[i][j][k] += dp[i + 1][j][k - 1];
                dp[i][j][k] += dp[i - 1][j][k - 1];
                dp[i][j][k] += dp[i][j + 1][k - 1];
                dp[i][j][k] += dp[i][j - 1][k - 1];
            }
        }
    }
}
```
Subtle differences between bottom-up and top-down

• Top-down requires significant stack space while bottom-up uses little.
• Top-down doesn’t support the memory saving trick.
• It is observed that when a problem has implicit simple DAG, it is faster and neater to code bottom-up.
• Bottom-up may not neatly compute only the necessary states. Sometimes redundant states are involved, resulting in additional computation time.

• Anyway, most of the time it is your call.
How to get a DP procedure

• Formulate its DAG (either explicitly or implicitly)
  – Determine the states (DAG nodes)
  – Determine the transitions (DAG edges)
  – Determine the order (implicit or explicit DAG topo-sort)

• Think about the computation costs
  – Affordable Memory
  – Affordable Time
Affordable Memory

• The maximum number of states during computation must fit in the memory limit
  – DP space most commonly is the number of total states. DAG size must not be too large.
  – Memory saving trick: reduce the memory requirement by one more dimension.

• Fibonacci Sequence:
  – We can only work up to Fib(N) where O(N) fits in the memory

• Grid Walking 2:
  – O(NMK) space, as there are N * M *K nodes in the DAG
  – or O(NM) if you use the memory saving trick
Memory saving trick

• Simply put: forget about the states if they will no longer be depended on.
  – In other words, let those states be gone forever after they are processed in the topo order.
Memory saving trick – Grid Walking 2

this layer will never be used again,

once we reach this layer
bool block[MAXN][MAXN];
int N, dp[2][MAXN][MAXN];
void walk() {
  int now = 0, pre;
  dp[now][1][1] = 1;
  for (int k = 1; k <= MAXK; k++) {
    pre = now; now ^= 1;
    memset(dp[now], 0, sizeof(dp[now]));
    for (int i = 1; i <= N; i++) {
      for (int j = 1; j <= N; j++) {
        if (block[i][j]) continue;
        dp[now][i][j] += dp[pre][i - 1][j];
        dp[now][i][j] += dp[pre][i + 1][j];
        dp[now][i][j] += dp[pre][i][j + 1];
        dp[now][i][j] += dp[pre][i][j - 1];
      }
    }
  }
}
Affordable Time

• The total computation cost must fit in the Time Limit.

• If the total number of states is $C$, then the time complexity would be $\Omega(C)$, as every state shall be computed.

• However, it could be more than that, depending on how much time is needed for each state.
Time needed to compute each state

• Recall the Cutting Sticks problem.
  – We have a complete DAG where each state \( \text{Cut}(i) \) depends on every \( \text{Cut}(j), \ j < i \)

• In order to compute \( \text{Cut}(i) \), we need to traverse \( i - 1 \) states. Therefore we need on average linear time to compute each state. The total time complexity is thus \( O(L^2) \).
## Online judge scenarios

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time</th>
<th>Space</th>
<th>Small</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci Sequence</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$N \leq 100$</td>
<td>$N \leq 10^7$</td>
</tr>
</tbody>
</table>
| Counting Problem   | $O(NS)$ | $O(S)$ | $N \leq 20$  | $N \leq 100$  
|                    |       |       | $S \leq 1000$ | $S \leq 10^6$ |
| Cutting Sticks     | $O(L^2)$ | $O(L)$ | $L \leq 100$ | $L \leq 5000$ |
| Grid Walking       | $O(NM)$ | $O(NM)$ | $N, M \leq 100$ | $N, M \leq 5000$ |
| Grid Walking 2     | $O(NMK)$ | $O(NM)$ | $N, M, K \leq 100$ | $N, M, K \leq 500$  
|                    |       |       | * Memory saving trick required * |
A DP problem typically contains important variables that can be used as a dimension of the DP state.
- Counting Problem: sum $S$ given
- Cutting Sticks: stick length $L$ given
- Grid Walking 2: number of steps $K$ given

More challenging DP questions demand more insights into the computation process, where we may have DP states corresponding not to the problem variables, but to our computation.
• Given N positive integers \( \{a_1, a_2 \ldots a_N\} \), as a multiset (allowing duplicates), split them into two multisets so that the two multisets have the smallest possible absolute difference.

• Example:
  - \( \{1,2,3,4\} \) Answer: \( \text{minDiff} = 0 \), e.g. \( \{1,4\}, \{2,3\} \)
  - \( \{2,4,5,6\} \) Answer: \( \text{minDiff} = 1 \), e.g. \( \{2,6\}, \{4,5\} \)
  - \( \{1,1,10\} \) Answer: \( \text{minDiff} = 8 \), e.g. \( \{1,1\}, \{10\} \)
DP formulation

• **State:** possible\((i, S)\), whether we can find a multiset of integers with sum \(S\), among the first \(i\) integers.

• Note that \(S\) is not directly given in the problem. But it is closely related to the balance that is asked for.
  
  – If we know possible\((N, S)\) is true, then we can split the integers into two multisets with \(S\) and \(S_{total} - S\), and their difference is \(|S - (S_{total} - S)|\).
  
  – The problem is equivalent to finding the value \(S\) closest to \(S_{total}/2\) so that possible\((N, S)\) is true.
Minimum Balance - DAG

\( a_1 = 3 \)

\( a_2 = 3 \)
Minimum Balance - DAG

(0, 0)  (0, \text{S}_{\text{total}})

Diagram showing the concept of minimum balance in a Directed Acyclic Graph (DAG) with nodes labeled (0, 0) and (0, S_{\text{total}}).
```c
bool dp[MAXN][MAXS];
int N, Stotal, a[MAXN]; // a[1..N] (1-based)

int minBalance()
{
    dp[0][0] = true;
    for (int i = 1; i <= N; i++) {
        for (int S = 0; S < MAXS; S++) {
            dp[i][S] |= dp[i - 1][S];
            if (S >= a[i])
                dp[i][S] |= dp[i - 1][S - a[i]];
        }
    }
    int ans = INF;
    for (int S = 0; S < MAXS; S++) {
        if (dp[N][S])
            ans = min(ans, abs(S - (Stotal - S)));
    }
    return ans;
}
```
Lucky Strings

• A string $X$ is lucky if it contains a lucky character $t$.
• Given the lucky character $t$, count how many lucky strings $X$ of length $N$ exist ($|X| = N$).
• Strings contain lowercase English letters. For simplicity, we only use the first 3 letters: ‘a’, ‘b’, ‘c’.
• **State:** count\( (i, \ exist) \)
  – \( i \): We have written \( i \) letters \( (1 \leq i \leq N) \)
  – \( \ exist \): we have already written at least once the lucky character c

• Note that \( \ exist \) is not directly retrieved from the variables defined by the problem. We come up with it for our computation.
Lucky character \( t = 'b' \)

- \( i = 0 \)
  - "", false
  - "a", false
    - "b", true
      - "ba", true
      - "bb", true
      - "bc", true
      - ...
    - "c", false
- \( i = 1 \)
  - "aa", false
  - "ab", true
    - "ba", true
  - "ac", false
- \( i = 2 \)
  - ...
  - ...
  - ...
  - ...
  - ...
  - ...

...
int N, dp[MAXN][2];

int count() {
    dp[0][0] = 1;
    for (int i = 1; i <= N; i++) {
        for (int exist = 0; exist < 2; exist++) {
            dp[i][1] += dp[i - 1][exist]; // write 'b'
            dp[i][exist] += 2 * dp[i - 1][exist]; // write 'a' or 'c'
        }
    }
    return dp[N][1];
}
A string $X$ is *lucky* if it contains a lucky substring $Y$.

Given the lucky substring $Y$, count how many lucky strings $X$ of length $N$ exist ($|X| = N$). $|Y| \leq N$.

All letters in $Y$ are distinct.
Lucky Strings 2 – DP Formulation

• **State:** count\((i, m)\)
  – \(i\): We have written \(i\) letters \((1 <= i <= N)\)
  – \(m\): the last \(m\) letters we wrote match the first \(m\) letters of \(Y\).
    • If \(m = |Y|\), then it means we have at least written \(Y\) once previously.

• Note that \(m\) is not directly retrieved from the variables defined by the problem either. We come up with it to track the substring matching process.
Lucky Strings 2 - DAG

Lucky substring $Y = \text{"ab"}$

- $i = 0$
  - “”, 0
  - “a”, 1
  - “b”, 0
  - “c”, 0

- $i = 1$
  - “a”, 1
  - “ab”, 2
  - “ac”, 0

- $i = 2$
  - “aa”, 1
  - “ab”, 2
  - “ac”, 0
  - “aba”, 2
  - “abb”, 2
  - “abc”, 2

- $i = 3$
  - “aaa”, 1
  - “aab”, 2
  - “aac”, 0
  - “aba”, 2
  - “abb”, 2
  - “abc”, 2
```c
int N, dp[MAXN][MAXN];
char Y[MAXN];

int count() {
    dp[0][0] = 1;
    int M = strlen(Y);
    for (int i = 1; i <= N; i++) {
        for (int m = 0; m < M; m++) {
            // write Y[m]
            dp[i][m + 1] += dp[i][m];
            // write anything other than Y[0] and Y[m], since we only
            // have 'a', 'b', 'c', and Y[m] != Y[0], there leaves
            // one choice left
            dp[i][0] += dp[i][m];
            // write Y[0]
            dp[i][1] += dp[i][m];
        }
        dp[i][M] += dp[i - 1][M] * 3; // anything works
    }
    return dp[N][M];
}```
• **Y has to contain distinct letters.** Otherwise our DP has to be changed.

• Consider $Y = \text{“abac”}$. When $i = 4$ and $m = 3$, if we write ‘b’, it would be a mismatch. But the new $m$ would be 2, instead of 0.

This is because the m letters we’ve just written could match some prefix of Y even when the next letter we are writing doesn’t match $Y[m]$.
Counting in DP – modulo arithmetic

• Answer could grow exponentially.

• \((a + b) \% M = ((a \% M) + (b \% M)) \% M\)
  – Ensure that \(\sim 2M\) fits in an 32-bit integer.
  – Typically when \(M \sim= 10^9\), the above is satisfied.

• \(a*b \% M = ((a \% M) * (b \% M)) \% M\)
  – Ensure \(M^2\) fits in an integer
  – Typically when \(M \sim= 10^9\), we must use 64-bit integer for the multiplication: \(((\text{long long})a * b) \% M\)
Summary

• Thinking about the DAG behind DP helps you
  – **Perceive** the problem more clearly
  – See how to **implement** the DP procedure neatly
  – **Determine** if a problem is solvable by DP, or how to solve it by DP.

• Formulate your DP
  – **Think about the DAG**: find states from problem variables, derived dimensions; get transitions
  – **Affordable memory**: check number of states, memory saving trick
  – **Affordable time**: check number of states, computation cost for each state
Exercises

• Cutting Sticks
  – Compute the maximum sum of values after cutting the original sticks into exactly $K$ segments.
  – Suppose the stick values are in dollars. Each cut costs you $D$ dollars. Find the maximum dollars you can achieve after cutting.

• Minimum Balance
  – Suppose $N$ is even. Find a split into two multisets so that not only the difference between two sums are minimum, but also the two multisets have the same number of integers.

• Lucky String
  – Analyze the complexity requirement for Lucky String and Lucky String 2.
  – Note that the time complexity for Lucky String 2 may additionally include the alphabet size, why?
• Lucky String 2:
  – Remove the constraint of Y’s letters being distinct.
  – Naive implementation of \textit{matchHead}(m, c) could work but would give larger time complexity. Use an efficient algorithm that performs \textit{matchHead}(m,c) in O(1) time.
  – Hint: string matching algorithm

• Grid Walking 2: What if \( N, M \leq 10, K \leq 10^9 \)?
  – We cannot afford any solution that needs \( O(10^9) \) time \textbf{OR} space.
  – Still solvable, as an advanced exercise 😊
  – Hint: why should \( N, M \) become smaller, i.e. \( \leq 10 \)?