# Dynamic Programming

## Methods in Computational Linguistic II

# Optimization

- Identify the three numbers that have the greatest product from the following set
- [5, 1, 0.5, 10, 2]

# Optimization

- Identify the three numbers that have the greatest product from the following set
- [-5, -1, -0.5, 10, 2]

# Optimization

- Optimization is the process of finding the "best" solution to some problem.
- There are many **valid** solutions
  - any set of 3 numbers is a solution.
  - Find the solution that has a maximum or minimum value
    - And do it efficiently
  - Search over the space of partial solutions

# Dynamic Programming

- The solution to a problem is made up of solutions to sub-problems.
- Subproblems are frequently reused.
- **Save the solution to subproblems**

# Fibbonacci

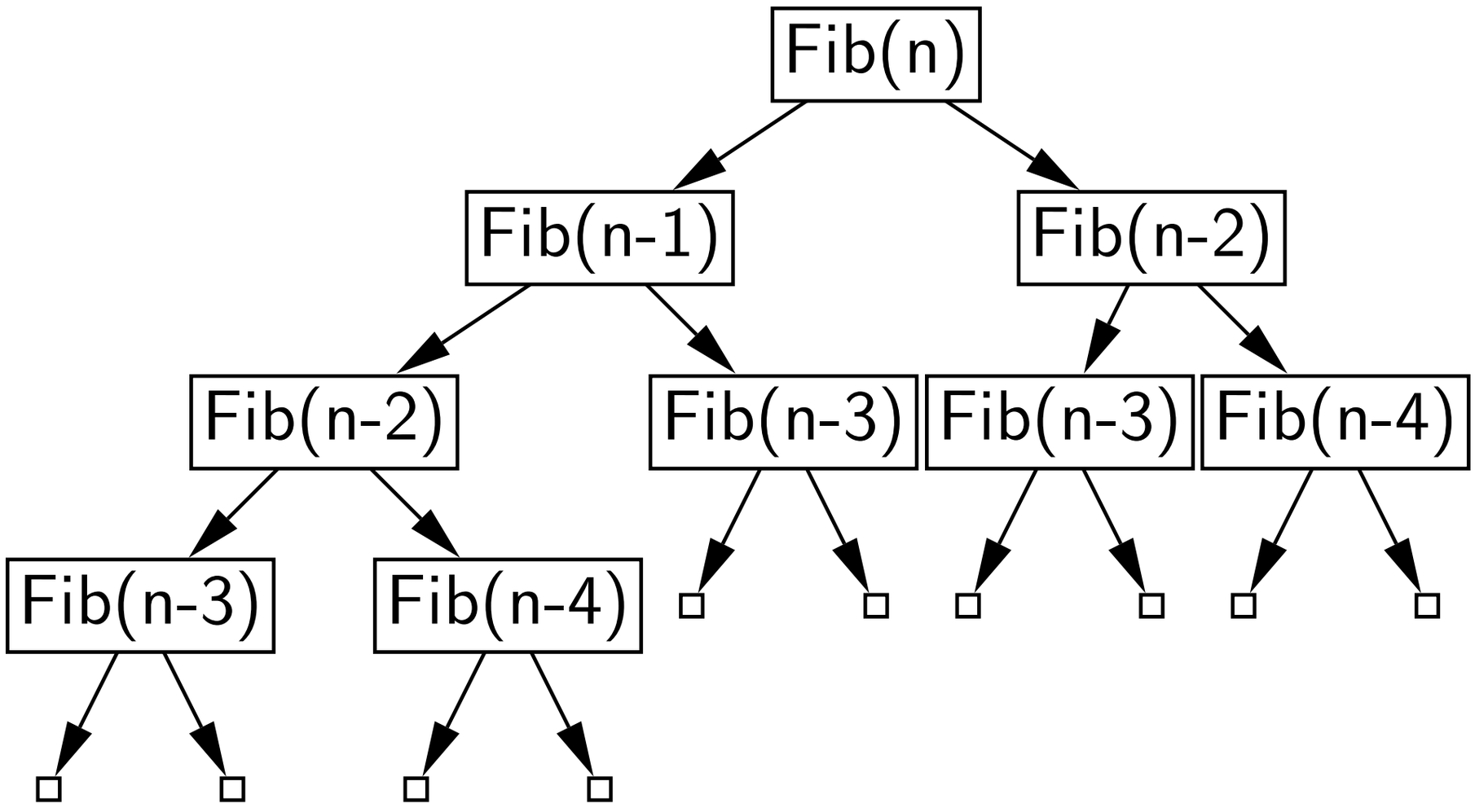
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, etc.

- Recursively
  - Fib(1) = 1
  - Fib(2) = 1
  - Fib(n) = Fib(n-1) + Fib(n-2)

# Fib(i)

```
def fib(n)
   if n == 1 or n == 2:
      return 1
   else:
      return fib(n-1) + fib(n-2)
```

# Why does this stink?




$1 = 2^0$

$2 = 2^1$

$4 = 2^2$

$8 = 2^3$

# Dynamic Programming Fib(i)

```
def fib(n)

  c = [0] * n
  c[1] = 1
  c[2] = 1
  for i in xrange(3, n+1):
    c[i] = c[i-1] + c[i-2]
  return c[n]
```

# Making Change

- Input: **n** denominations of coins.
- Problem: Make change for an amount of money **C** using as few coins as possible
- All values of coins and **C** are positive integers.

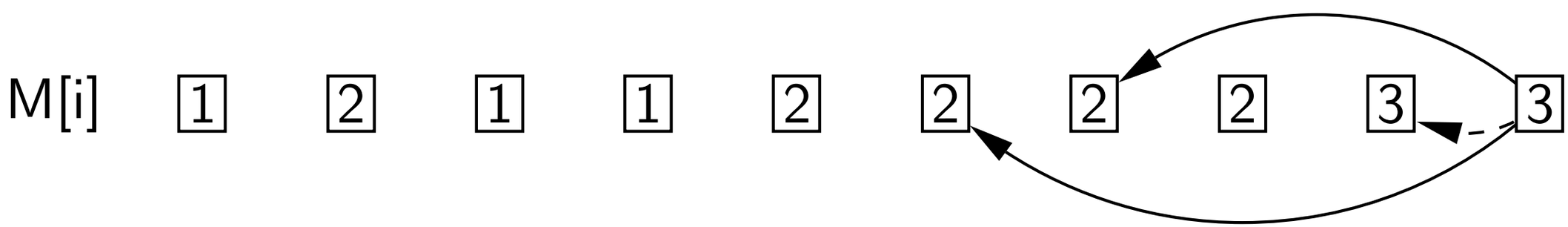
# Making change

- **M(j)** = the minimum number of coins required to make change for some amount **j**
- **M(j) = min { M(j - v[i]) } + 1**
- The smallest number of coins that could make **j** is the smallest number required to make **j-v[i]** plus one.

- To calculate: start from the smallest amount and build up a table, **M**

# Making change

- v = [1, 3, 4]
- C = 10



# Making Change

```
def minChange(C, v)

  M = [float("inf")] * C
  for j in xrange(1, C + 1):

    min = float("inf")
    for i in xrange(len(v)):
      if M[j-v[i]] + 1 < min:
        min = M[j-v[i]] + 1
return M[C]
```

# Elements of Dynamic Programming

- Optimal Substructure
- Overlapping Subproblems
- Reconstruction of an Optimal Solution

# Optimal Substructure

- A problem demonstrates **optimal substructure** if an optimal solution is comprised of optimal solutions to subproblems.
- Identifying optimal substructures
  - Show that the solution involves making a **choice** leaving a set of subproblems to be solved.
  - Suppose that you are given **a choice** that leads to an optimal solution.
  - Determine which subproblems remain after this optimal choice, and how to characterize the space of subproblems.
  - Show that the solutions to subproblems used in the optimal solution must also be optimal

# Overlapping Subproblems

- Dynamic Programming leads to efficient solutions to problems when subproblems are frequently **reused**
- Efficiency is introduced by saving the results of subproblems.
- In practice the solutions are stored in a table, known as a Dynamic Programming Table, or DP Table.

# Reconstruction of an Optimal Solution

- The DP Table contains the value that is being optimized.
- Reconstructing the set of choices made is not always trivial.
- This typically involves **storing the choices** made in a second DP table of identical dimensions.
    - Other implementations store the choice in the original table in a tuple or other data structure

# Minimum Edit Distance

- How different are the strings **LEAD** and **LAST**?

# Minimum Edit Distance

- How different are **LEAD** and **LAST**?
- Approach: Count the number of editing operations required to transform one to the other.

# Minimum Edit Distance

- How different are **LEAD** and **LAST**?
- Approach: Count the number of editing operations required to transform one to the other.
- Define three (or two) edit operations:
  - Insert, Delete and Substitute.
- This is known as **Minimum Edit Distance** (MED) or **Levenshtein Distance**

# Minimum Edit Distance

- How different are **LEAD** and **LAST**?
- Three.

- One deletion: **LEAD → LAD**
- One substitution: **LAD → LAT**
- One insertion: **LAT → LAST**

# Minimum Edit Distance

- Optimal substructure.
- Two strings **s** and **t**.
- **s[:i]** and **t[:j]** have a minimum edit distance of **c**
- **d(s[:i], t[:j]) = c**
- Claim: **c** is the smallest of four values.
  - **d(s[:i-1], t[:j-1])** if **s[i] == t[j]**
  - **d(s[:i-1], t[:j-1]) + 1** otherwise (substitution)
  - **d(s[:i], t[:j-1]) + 1** (insertion)
  - **d(s[:i-1], t[:j]) + 1** (deletion)

# Recursive MED

```
def med(s, t):
  if len(s) == 0 and len(t) == 0:
    return 0
  if len(s) == 0:
    return med(s, t[:-1]) + 1
  if len(t) == 0:
    return med(s[:-1], t) + 1
sub = 1
if s[-1] == t[-1]:
  sub = 0
return min( med(s, t[:-1]) + 1,
            med(s[:-1], t]) + 1,
            med(s[:-1], t[:-1]) + sub)
```

# Dynamic Programming for MED

- Rather than use recursion starting with full strings, **build up** a solution to the final problem from subproblems.

- Construct a table
  - let **k = len(s)** and **l = len(t)**
  - Goal: construct a **k+1** by **l+1** table **M** such that **M[i][j]** is the minimum number of edits to convert **s[:i]** to **t[:j]**
  - When this goal is accomplished
    **d(s,t) = M[k+1][l+1]**

# Example MED Table

| | | ∅ | L | A | S | T |
|---|---|---|---|---|---|---|
| | ∅ | | | | | |
| | L | | | | | |
| | E | | | | | |
| | A | | | | | |
| | D | | | | | |

| | ∅ | L | A | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | | | | |
| E | 2 | | | | |
| A | 3 | | | | |
| D | 4 | | | | |

| | ∅ | **L** | A | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| **L** | 1 | | | | |
| E | 2 | | | | |
| A | 3 | | | | |
| D | 4 | | | | |

| | ∅ | **L** | A | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| **L** | 1 | 0 | | | |
| E | 2 | | | | |
| A | 3 | | | | |
| D | 4 | | | | |

| | ∅ | **L** | A | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | | | |
| **E** | 2 | | | | |
| A | 3 | | | | |
| D | 4 | | | | |

| | ∅ | **L** | A | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | | | |
| **E** | 2 | 1 | | | |
| A | 3 | | | | |
| D | 4 | | | | |

| | ∅ | **L** | A | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | | | |
| E | 2 | 1 | | | |
| **A** | 3 | 2 | | | |
| D | 4 | | | | |

| | ∅ | **L** | A | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | | | |
| E | 2 | 1 | | | |
| A | 3 | 2 | | | |
| **D** | 4 | 3 | | | |

| | ∅ | L | **A** | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| **L** | 1 | 0 | 1 | | |
| E | 2 | 1 | | | |
| A | 3 | 2 | | | |
| D | 4 | 3 | | | |

| | ∅ | L | **A** | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | | |
| **E** | 2 | 1 | 1 | | |
| A | 3 | 2 | | | |
| D | 4 | 3 | | | |

| | ∅ | L | **A** | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | | |
| E | 2 | 1 | 1 | | |
| **A** | 3 | 2 | 1 | | |
| D | 4 | 3 | | | |

| | ∅ | L | **A** | S | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | | |
| E | 2 | 1 | 1 | | |
| A | 3 | 2 | 1 | | |
| **D** | 4 | 3 | 2 | | |

| | ∅ | L | A | **S** | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| **L** | 1 | 0 | 1 | 2 | |
| E | 2 | 1 | 1 | | |
| A | 3 | 2 | 1 | | |
| D | 4 | 3 | 2 | | |

| | ∅ | L | A | **S** | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | 2 | |
| **E** | 2 | 1 | 1 | 2 | |
| A | 3 | 2 | 1 | | |
| D | 4 | 3 | 2 | | |

| | ∅ | L | A | **S** | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | 2 | |
| E | 2 | 1 | 1 | 2 | |
| **A** | 3 | 2 | 1 | 2 | |
| D | 4 | 3 | 2 | | |

| | ∅ | L | A | **S** | T |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | 2 | |
| E | 2 | 1 | 1 | 2 | |
| A | 3 | 2 | 1 | 2 | |
| **D** | 4 | 3 | 2 | 2 | |

| | ∅ | L | A | S | **T** |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| **L** | 1 | 0 | 1 | 2 | 3 |
| E | 2 | 1 | 1 | 2 | |
| A | 3 | 2 | 1 | 2 | |
| D | 4 | 3 | 2 | 2 | |

| | ∅ | L | A | S | **T** |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | 2 | 3 |
| **E** | 2 | 1 | 1 | 2 | 3 |
| A | 3 | 2 | 1 | 2 | |
| D | 4 | 3 | 2 | 2 | |

| | ∅ | L | A | S | **T** |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | 2 | 3 |
| E | 2 | 1 | 1 | 2 | 3 |
| **A** | 3 | 2 | 1 | 2 | 3 |
| D | 4 | 3 | 2 | 2 | |

| | ∅ | L | A | S | **T** |
|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 |
| L | 1 | 0 | 1 | 2 | 3 |
| E | 2 | 1 | 1 | 2 | 3 |
| A | 3 | 2 | 1 | 2 | 3 |
| **D** | 4 | 3 | 2 | 2 | 3 |

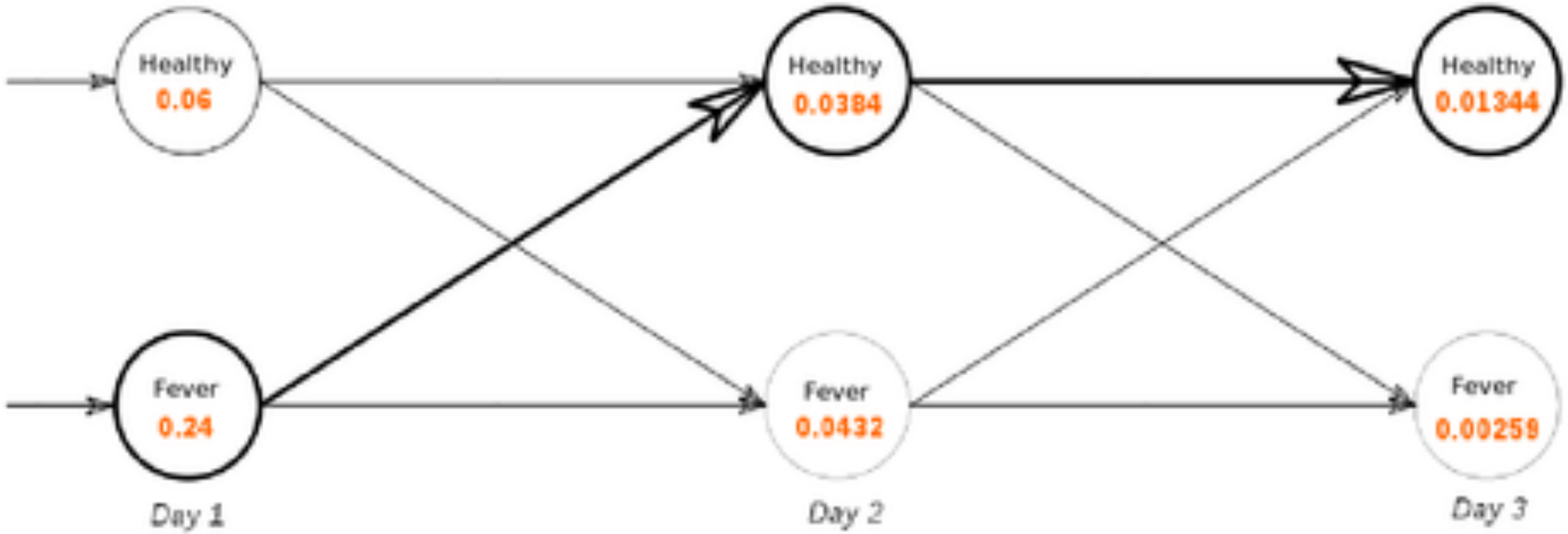
# Programming time

- Let's code up Minimum Edit Distance.

# Using MED for Alignment

- What about aligning non-equal tokens, like, say phones to orthographic tokens?
- Need to define a **cost** function for substitution.
- This generalizes MED to allow for variable cost insert, delete and substitute operations.

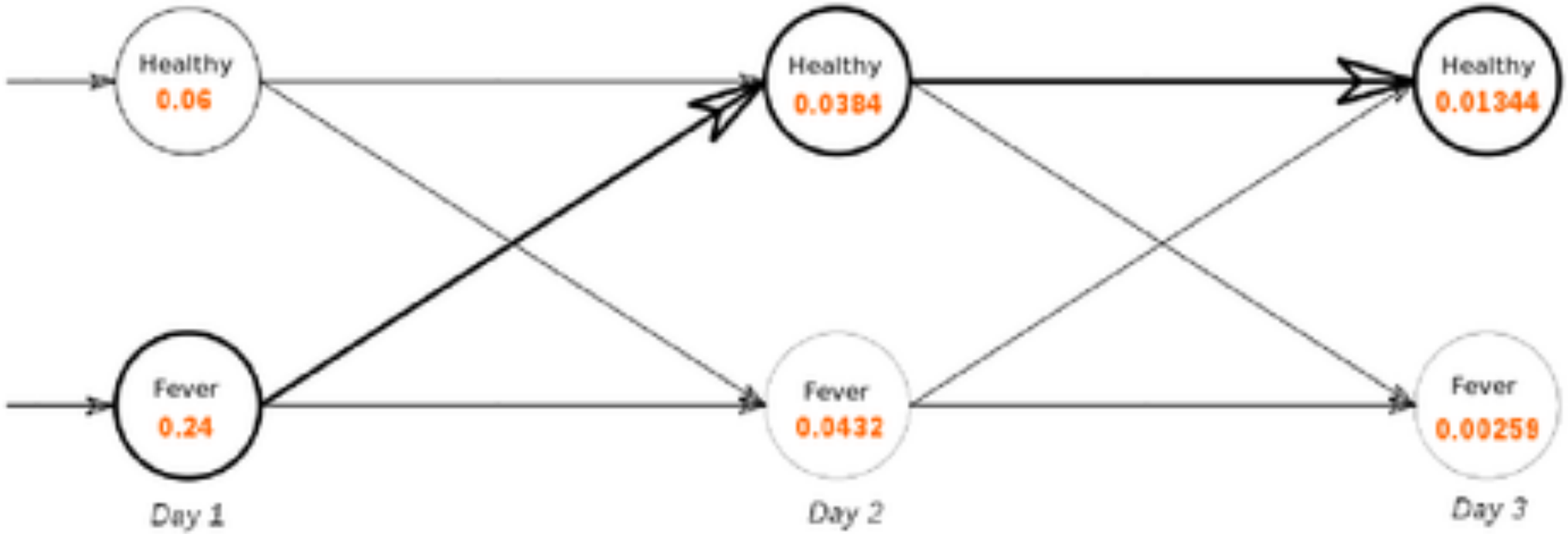
# Viterbi Algorithm




- At its most concise:

$$
\begin{aligned}
V_{0,k} &= \mathrm{P}\left(y_0 \mid k\right) \cdot \pi_k \\
V_{t,k} &= \mathrm{P}\left(y_t \mid k\right) \cdot \max_{x \in S} \left(a_{x,k} \cdot V_{t-1,x}\right)
\end{aligned}
$$

# Viterbi Algorithm




$$
\begin{aligned}
V_{0,k} &= \mathrm{P}(y_0 \mid k) \cdot \pi_k \\
V_{t,k} &= \mathrm{P}(y_t \mid k) \cdot \max_{x \in S} \left( a_{x,k} \cdot V_{t-1,x} \right)
\end{aligned}
$$

- Build up V[0,k] for all values of k.
- Then repeat for all values of t.

- Next time:
- Segmentation