Dynamic Programming: Change Problem

Pavel Pevzner

Department of Computer Science and Engineering
University of California, San Diego

Data Structures and Algorithms
Algorithmic Toolbox

Outline

Change problem

Find the minimum number of coins needed to make change.

Formally

Change problem

Input: An integer *money* and positive integers $coin_1, \ldots, coin_d$.

Output: The minimum number of coins with denominations $coin_1, \ldots, coin_d$ that changes *money*.

Greedy Way

```
GreedyChange(money)
Change ← empty collection of coins
while money > 0:
    coin ← largest denomination
           that does not exceed money
    add coin to Change
    money ← money − coin
return Change
```

Changing Money

in the US

40 cents $= 25 + 10 + 5$

Greedy

Changing Money

in Tanzania

40 cents $=$ $25 + 10 + 5$ $=$ $20 + 20$

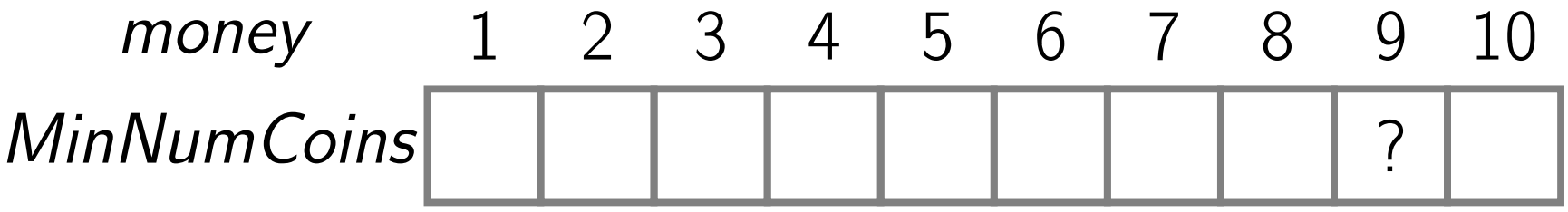
Greedy is not Optimal

Outline

Recursive Change

Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?

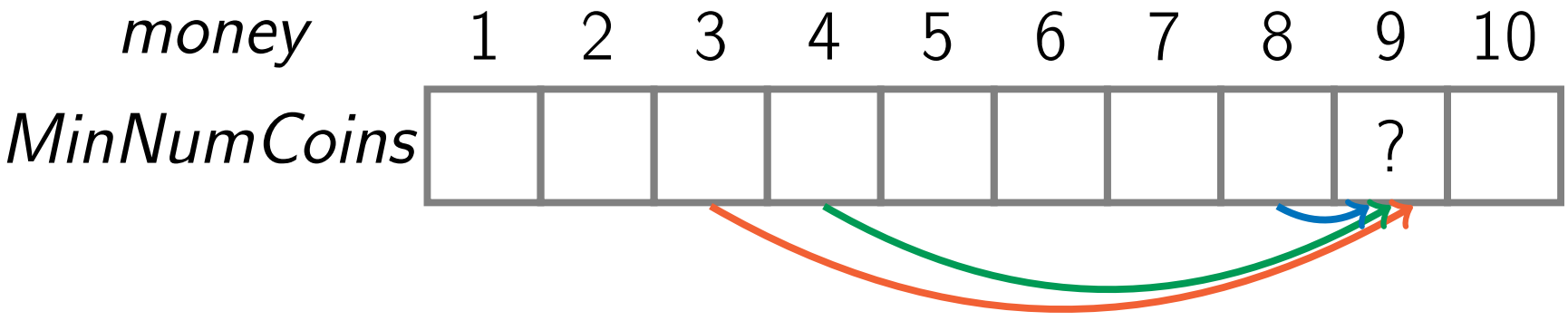
$$MinNumCoins(9) = \min \begin{cases} MinNumCoins(9-6)+1 \\ MinNumCoins(9-5)+1 \\ MinNumCoins(9-1)+1 \end{cases}$$



Recursive Change

Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?

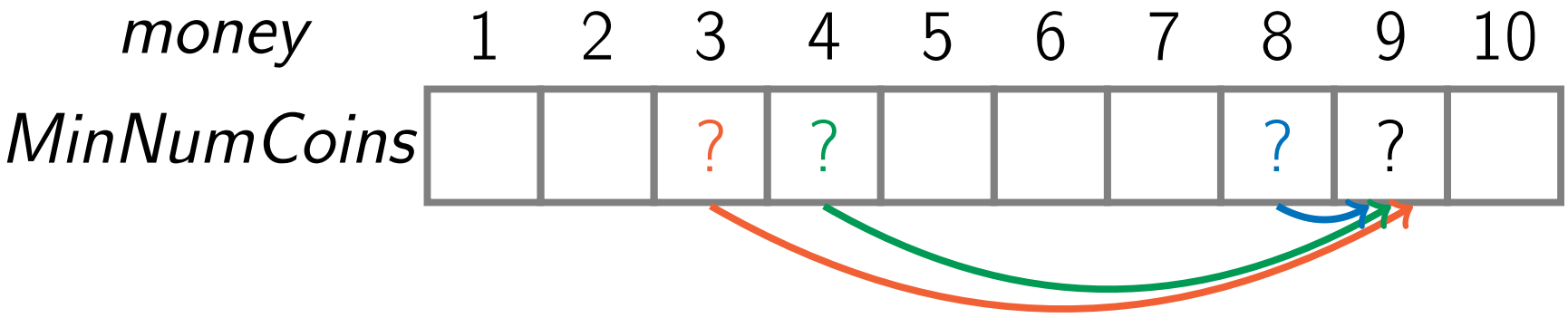
$$MinNumCoins(9) = \min \begin{cases} MinNumCoins(3) + 1 \\ MinNumCoins(4) + 1 \\ MinNumCoins(8) + 1 \end{cases}$$



Recursive Change

Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?

$$MinNumCoins(9) = \min \begin{cases} MinNumCoins(3) + 1 \\ MinNumCoins(4) + 1 \\ MinNumCoins(8) + 1 \end{cases}$$



Recurrence for Change Problem

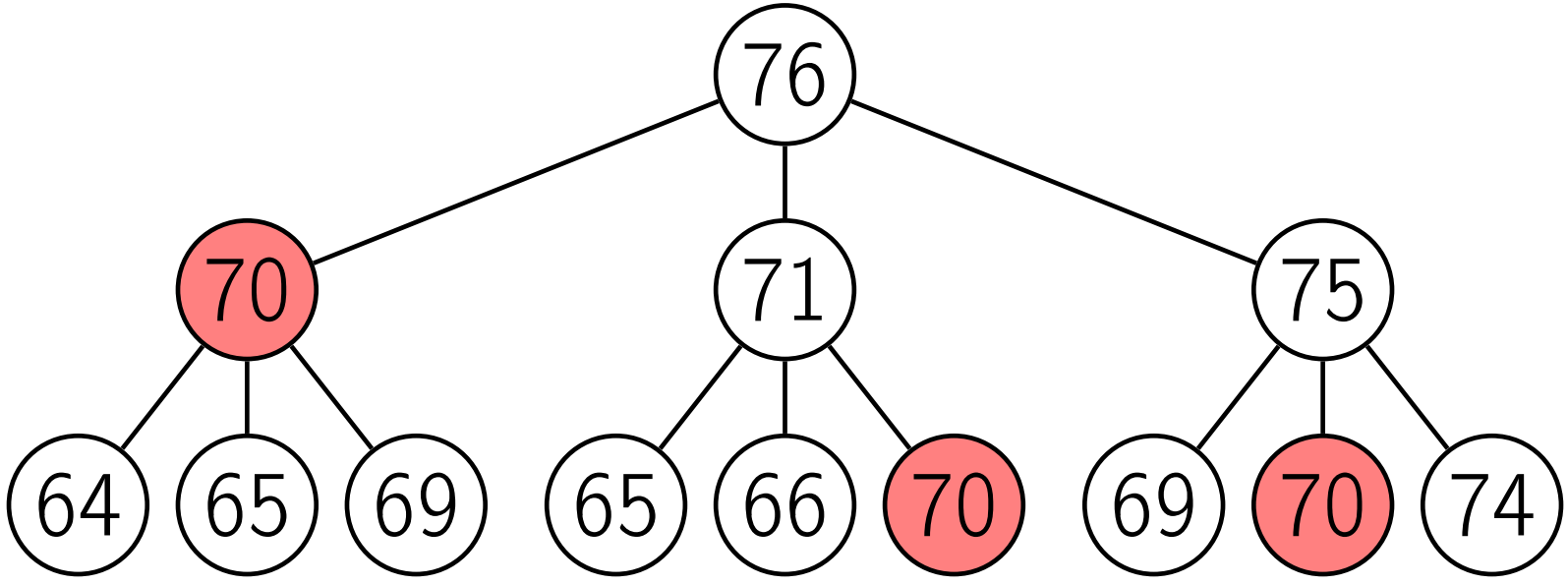
$$MinNumCoins(money) =$$

$$\min \begin{cases} MinNumCoins(money - coin_1) + 1 \\ MinNumCoins(money - coin_2) + 1 \\ \cdots \\ MinNumCoins(money - coin_d) + 1 \end{cases}$$

RecursiveChange($money$, $coins$)

```
if money = 0:
  return 0
MinNumCoins ← ∞
for i from 1 to |coins|:
  if money ≥ coin_i:
    NumCoins ← RecursiveChange(money − coin_i, coins)
    if NumCoins + 1 < MinNumCoins:
      MinNumCoins ← NumCoins + 1
return MinNumCoins
```

How Fast is `RecursiveChange`?



the optimal coin combination for 70 cents is computed at least **three** times!

the optimal coin combination for 30 cents is computed **trillions** of times!

Hint

Wouldn't it be nice to know all the answers for changing $money - coin_i$ by the time we need to compute an optimal way of changing *money*?

Instead of the time-consuming calls to

`RecursiveChange`$(money - coin_i, coins)$

we would simply look up these values!

Outline

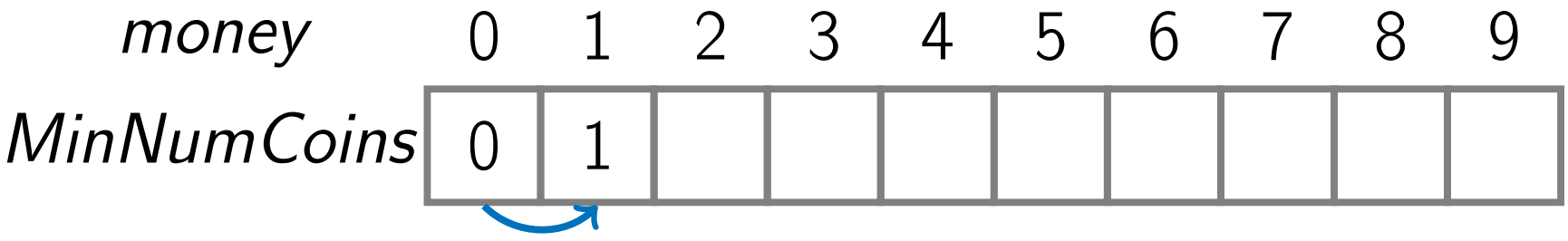
Dynamic Programming

What is the minimum number of coins needed to change 0 cents for denominations 6, 5, and 1?

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0									

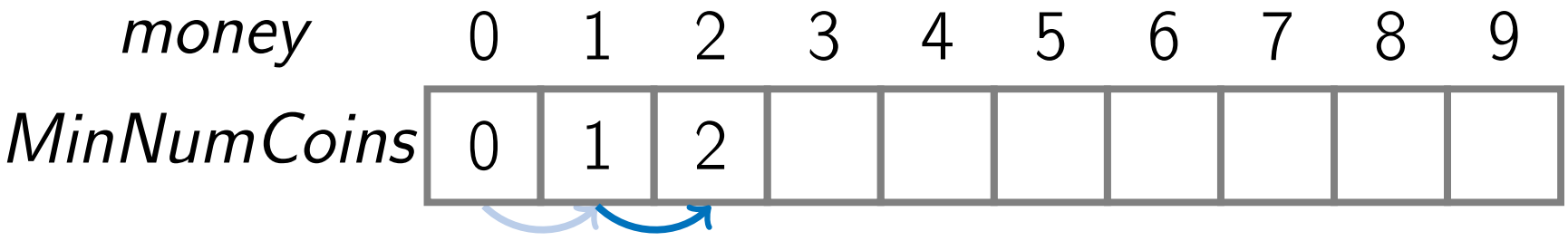
Dynamic Programming

What is the minimum number of coins needed to change 1 cent for denominations 6, 5, and 1?



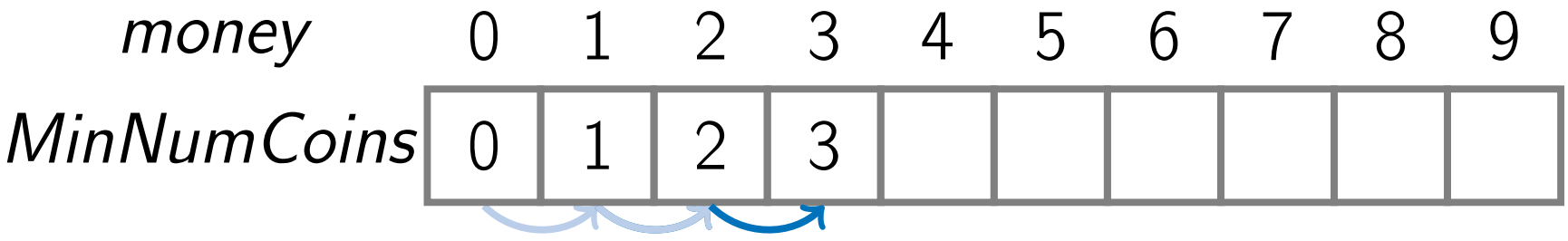
Dynamic Programming

What is the minimum number of coins needed to change 2 cents for denominations 6, 5, and 1?



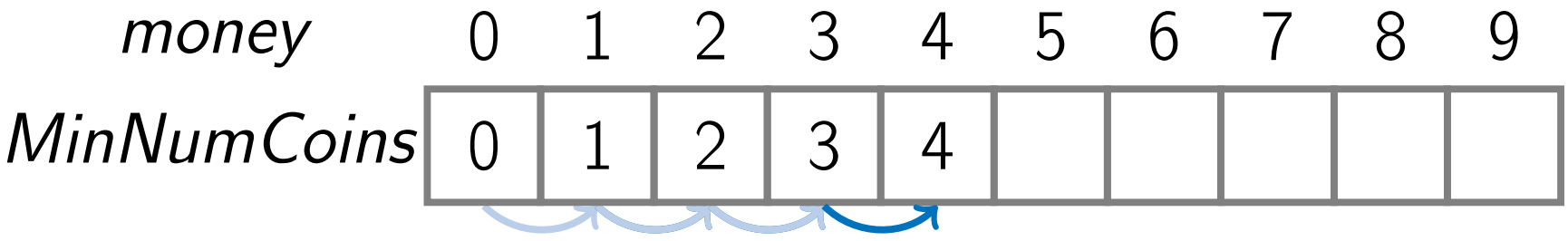
Dynamic Programming

What is the minimum number of coins needed to change 3 cents for denominations 6, 5, and 1?



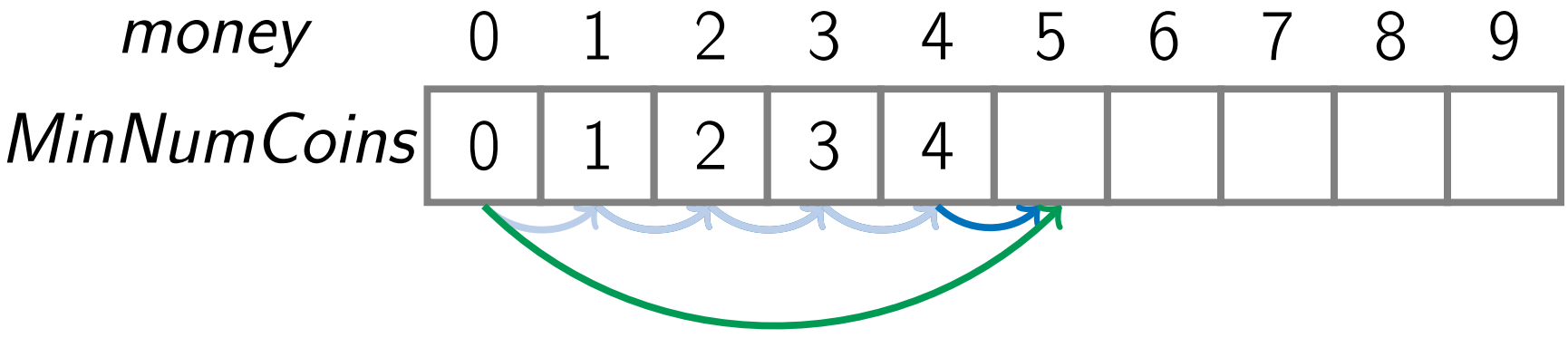
Dynamic Programming

What is the minimum number of coins needed to change 4 cents for denominations 6, 5, and 1?



Dynamic Programming

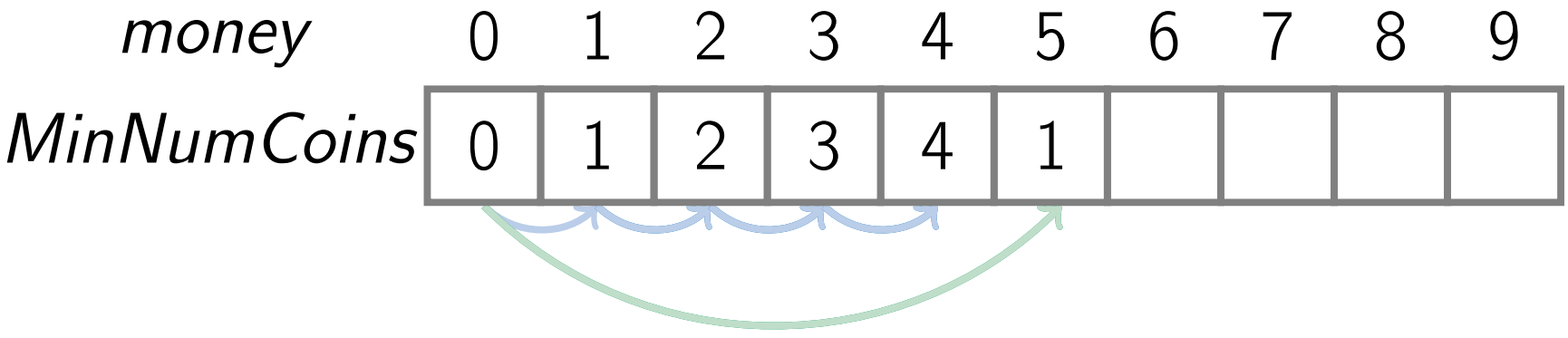
What is the minimum number of coins needed to change 5 cents for denominations 6, 5, and 1?



$$\min \begin{cases} MinNumCoins(0) + 1 \\ MinNumCoins(4) + 1 \end{cases}$$

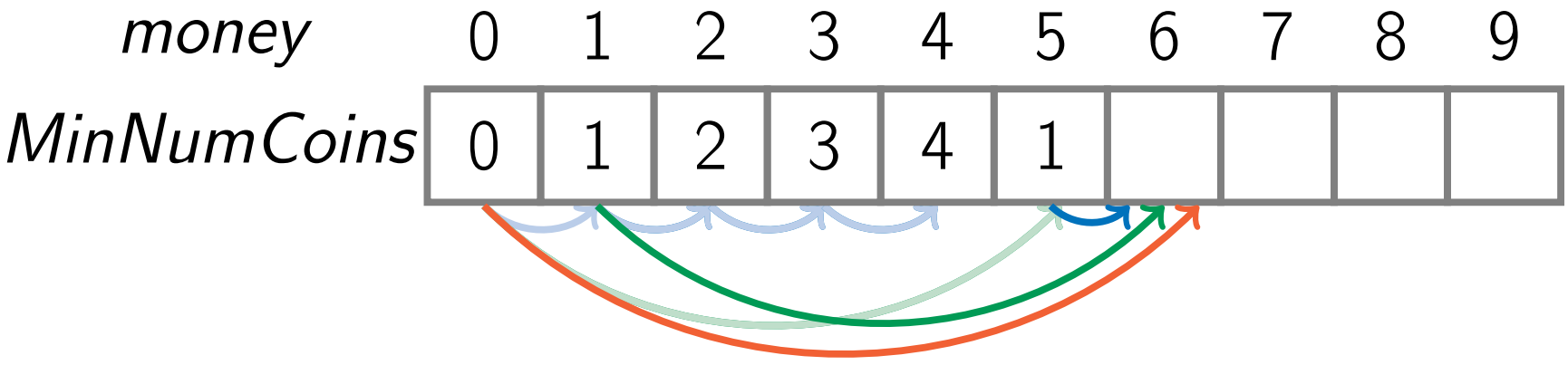
Dynamic Programming

What is the minimum number of coins needed to change 5 cents for denominations 6, 5, and 1?



Dynamic Programming

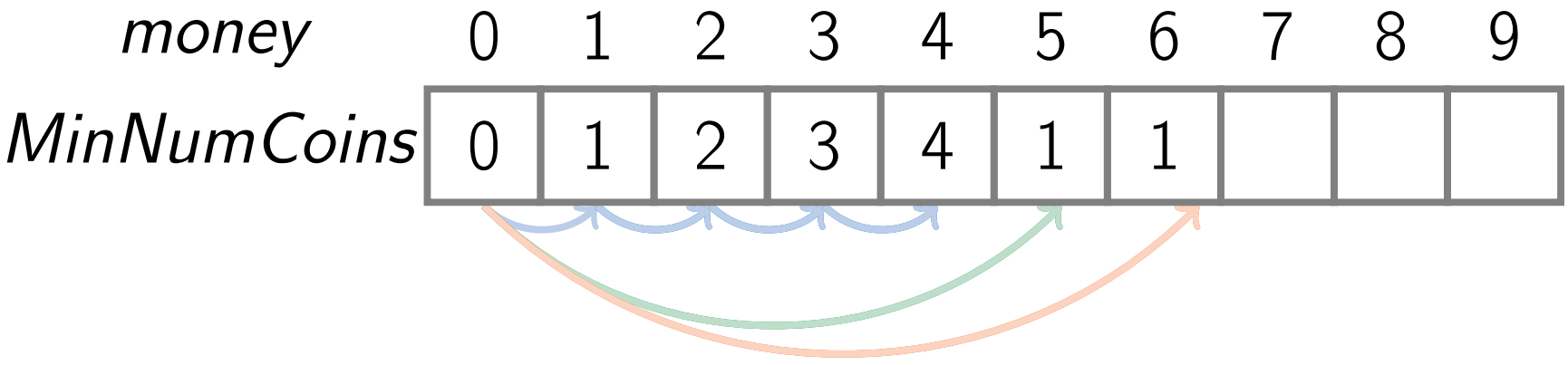
What is the minimum number of coins needed to change 6 cents for denominations 6, 5, and 1?



$$\min \begin{cases} MinNumCoins(0) + 1 \\ MinNumCoins(1) + 1 \\ MinNumCoins(5) + 1 \end{cases}$$

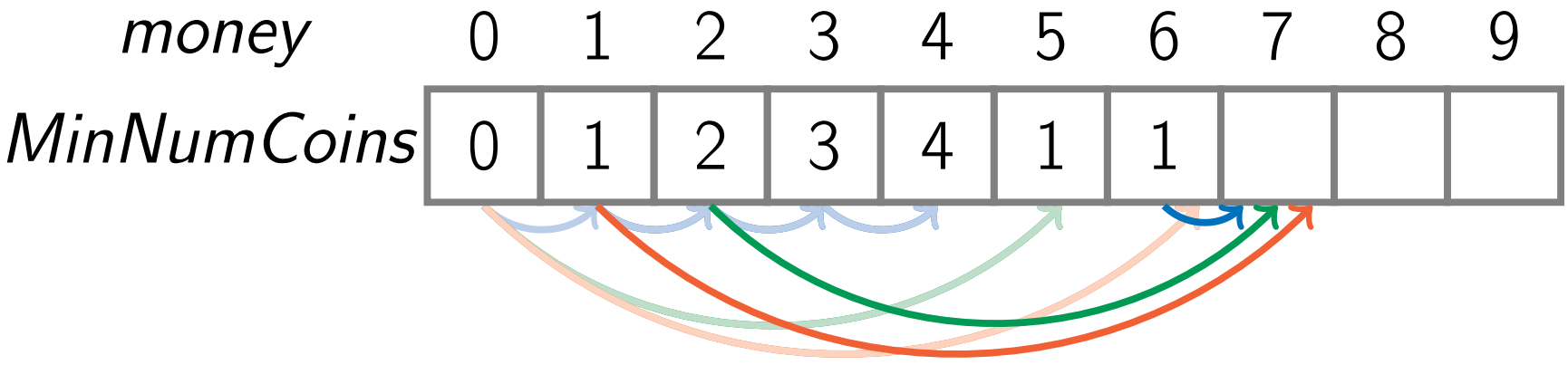
Dynamic Programming

What is the minimum number of coins needed to change 6 cents for denominations 6, 5, and 1?



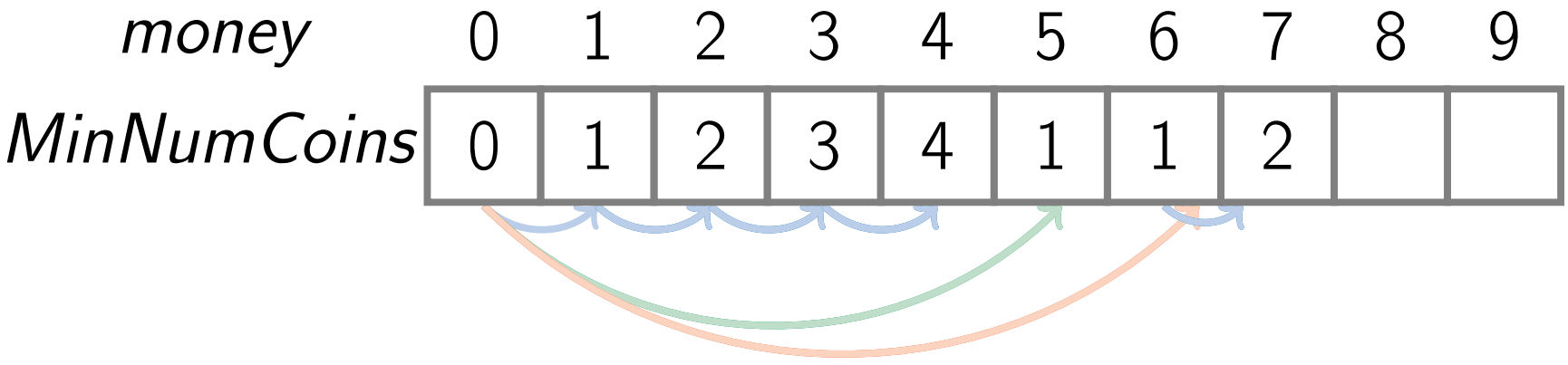
Dynamic Programming

What is the minimum number of coins needed to change 7 cents for denominations 6, 5, and 1?



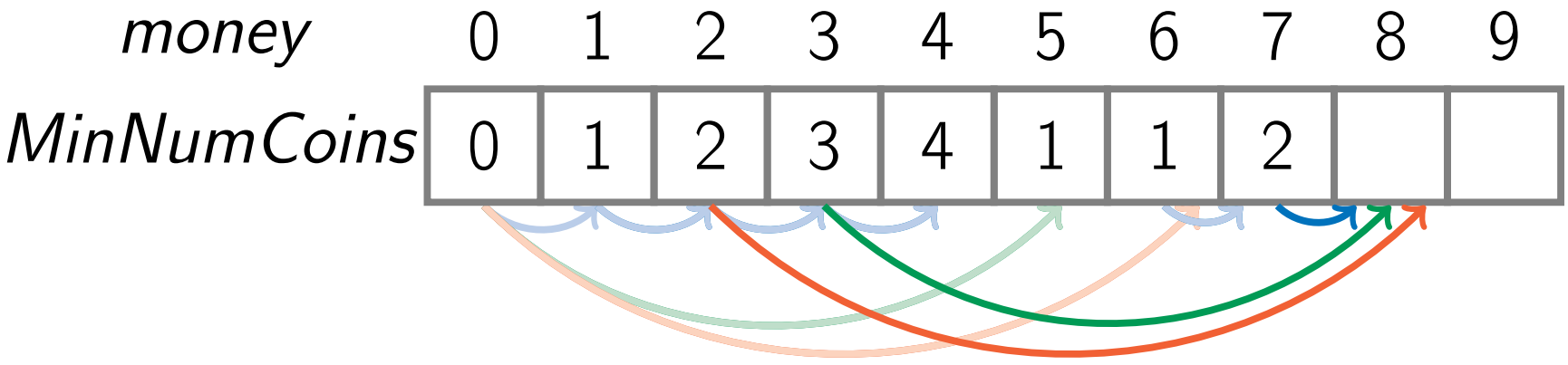
Dynamic Programming

What is the minimum number of coins needed to change 7 cents for denominations 6, 5, and 1?



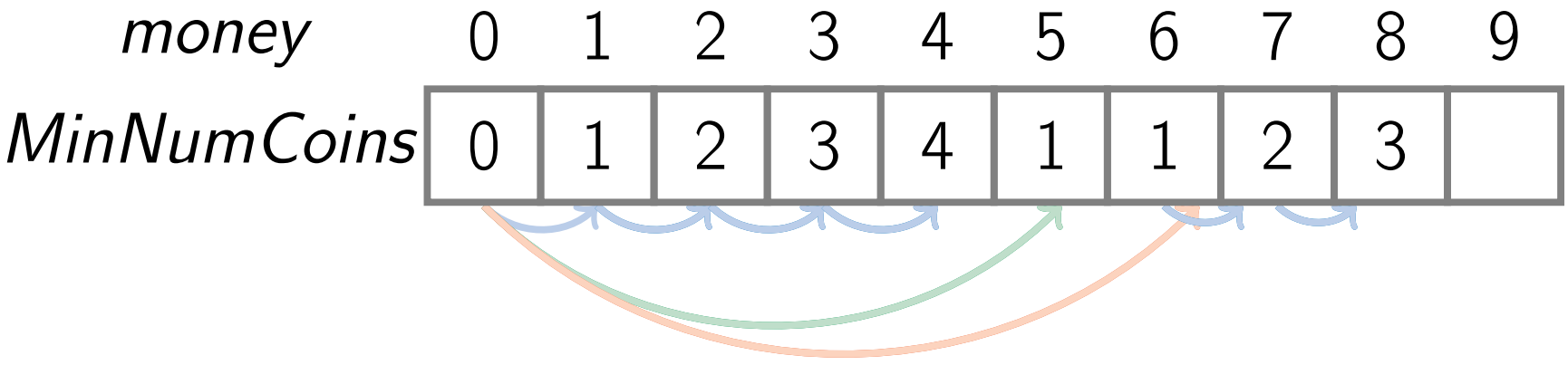
Dynamic Programming

What is the minimum number of coins needed to change 8 cents for denominations 6, 5, and 1?



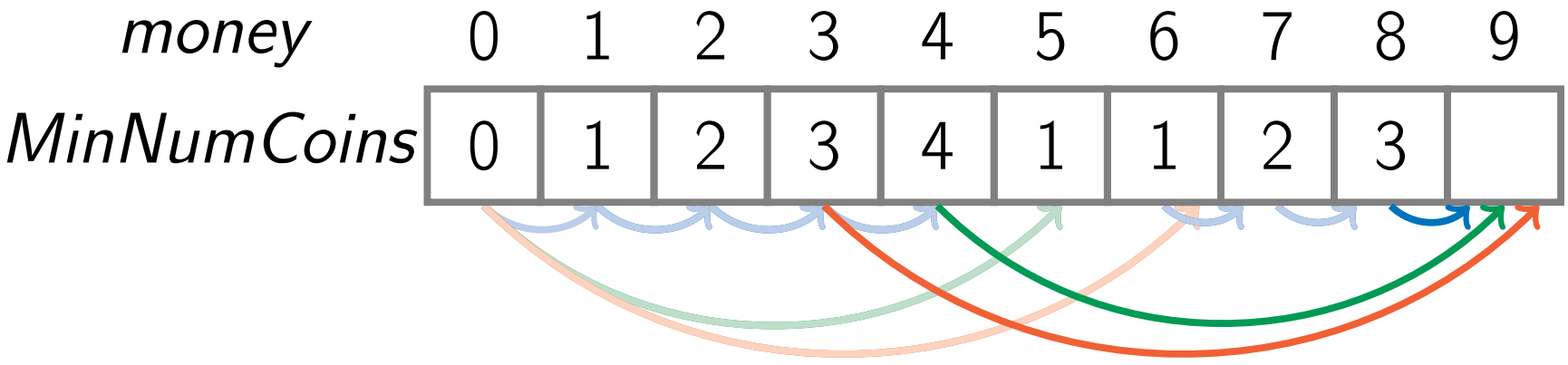
Dynamic Programming

What is the minimum number of coins needed to change 8 cents for denominations 6, 5, and 1?



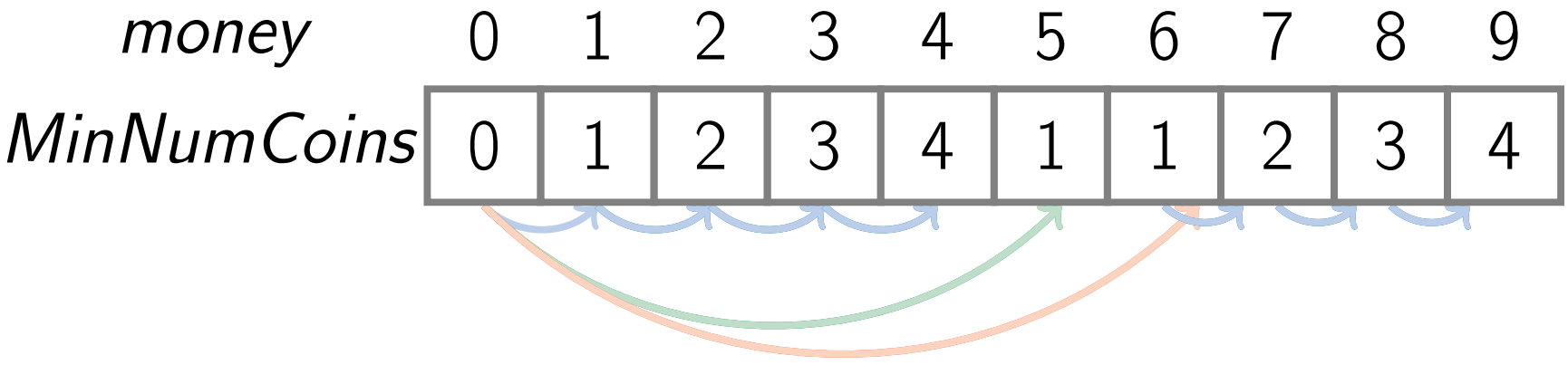
Dynamic Programming

What is the minimum number of coins needed to change 9 cents for denominations 6, 5, and 1?



Dynamic Programming

What is the minimum number of coins needed to change 9 cents for denominations 6, 5, and 1?



DPChange($money$, $coins$)

$MinNumCoins(0) \leftarrow 0$
`for` m `from 1 to` $money$:
  $MinNumCoins(m) \leftarrow \infty$
  `for` i `from 1 to` $|coins|$:
    `if` $m \geq coin_i$:
      $NumCoins \leftarrow MinNumCoins(m - coin_i) + 1$
      `if` $NumCoins < MinNumCoins(m)$:
        $MinNumCoins(m) \leftarrow NumCoins$
`return` $MinNumCoins(money)$

"Programming" in "Dynamic Programming" Has Nothing to Do with Programming!

Richard Bellman developed this idea in 1950s working on an Air Force project.

At that time, his approach seemed completely impractical.

He wanted to hide that he is really doing math from the Secretary of Defense.

Richard Bellman

...What name could I choose? I was interested in planning but planning, is not a good word for various reasons. I decided therefore to use the word, "programming" and I wanted to get across the idea that this was dynamic. **It was something not even a Congressman could object to.** So I used it as an umbrella for my activities.

Dynamic Programming: String Comparison

Pavel Pevzner

Department of Computer Science and Engineering
University of California, San Diego

Algorithmic Toolbox
Data Structures and Algorithms

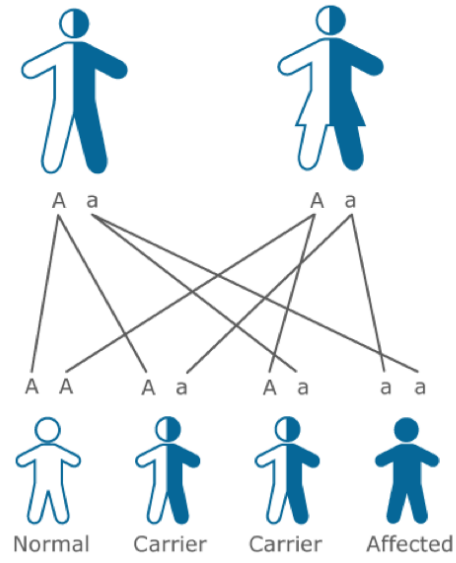
Cystic Fibrosis

Cystic fibrosis (CF): An often fatal disease which affects the respiratory system and produces an abnormally large amount of mucus.

- Mucus is a slimy material that coats epithelial surfaces and is secreted into fluids such as saliva.

Approximately 1 in 25 Humans Carry a Faulty CF Gene

- When BOTH parent carry a faulty gene, there is a 25% chance that their child will have cystic fibrosis.
- In the early 1980s biologists hypothesized that CF is caused by mutations in an unidentified gene.



Where Is the Cystic Fibrosis Gene?

- In the late 1980s, biologists narrowed the search for the CF gene to a million nucleotide long region on chromosome 7.
- However, this regions contained many genes and it was unclear which of them was responsible for CF.

chromosome 7

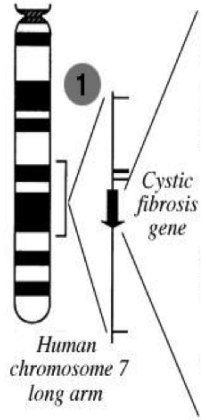
Hint 1: Cystic fibrosis involves sweet secretion with abnormally high sodium levels

Hint 2: By that time biologists already knew the sequences of some genes responsible for secretion, e.g., **ATP binding proteins** act as transport channels responsible for secretion

Hint 3: Should we search for genes in this region that are similar to known genes responsible for secretion?

Identifying the Cystic Fibrosis Gene

- BINGO: One of the genes in this region was similar to **ATP binding proteins** that act as transport channels responsible for secretion.



Hint 1: Cystic fibrosis involves sweet secretion with abnormally high sodium levels

Hint 2: By that time biologists already knew the sequences of some genes responsible for secretion, e.g., **ATP binding proteins** act as transport channels responsible for secretion

Hint 3: Should we search for genes in this region that are similar to known genes responsible for secretion?

Outline

The Alignment Game

```
A T G T T A T A
A T C G T C C
```

Alignment game: remove all symbols from two strings in such a way that the number of points is maximized:

Remove the 1st symbol from **both** strings:

- 1 point if the symbols match,
- 0 points if they don't match

Remove the 1st symbol from **one** of the strings:

- 0 points

The Alignment Game

A	T	G	T	T	A	T	A
A	T	C	G	T	C	C	
+1							

The Alignment Game

A	T	G	T	T	A	T	A
A	T	C	G	T	C	C	
+1	+1						

The Alignment Game

```
A  T  -  G  T  T  A  T  A
A  T  C  G  T  C  C
+1 +1
```

The Alignment Game

```
A  T  -  G  T  T  A  T  A
A  T  C  G  T  C  C
+1 +1    +1
```

The Alignment Game

```
A T - G T T A T A
A T C G T C C
+1 +1   +1 +1
```

The Alignment Game

```
A T - G T T A T A
A T C G T - C C
+1 +1   +1 +1
```

The Alignment Game

```
A T - G T T A T A
A T C G T - C C
+1 +1   +1 +1
```

The Alignment Game

```
A T - G T T A T A
A T C G T - C - C
+1 +1   +1 +1
```

The Alignment Game

```
A T - G T T A T A
A T C G T - C - C
+1 +1   +1 +1
```

The Alignment Game

```
A T - G T T A T A
A T C G T - C - C
+1 +1   +1 +1       =4
```

Sequence Alignment

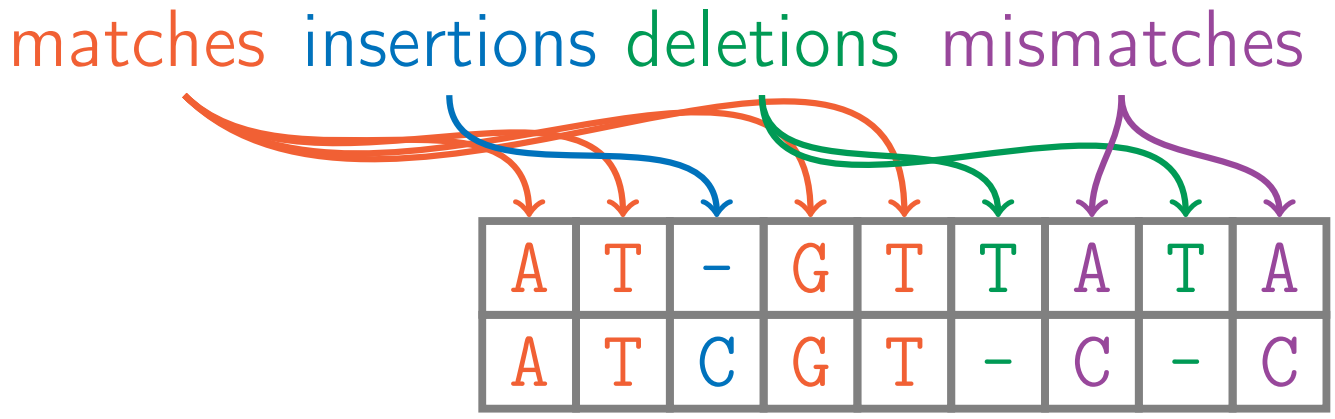
A	T	-	G	T	T	A	T	A
A	T	C	G	T	-	C	-	C

Alignment of two strings is a two-row matrix:

1st row: symbols of the 1st string (in order) interspersed by "–"

2nd row: symbols of the 2nd string (in order) interspersed by "–"

Sequence Alignment



Alignment Score

A	T	-	G	T	T	A	T	A
A	T	C	G	T	-	C	-	C

+1 +1 -1 +1 +1 -1 +0 -1 +0 =1

Alignment score: premium for every match ($+1$) and penalty for every mismatch ($-\mu$), indel ($-\sigma$).

Example: $\mu = 0$ and $\sigma = 1$

Alignment Score

$$\#\text{matches} - \mu \cdot \#\text{mismatches} - \sigma \cdot \#\text{indels}$$

Optimal alignment

Input: Two strings, mismatch penalty μ, and indel penalty σ.

Output: An alignment of the strings maximizing the score.

Common Subsequence

A	T	-	G	T	T	A	T	A
A	T	C	G	T	-	C	-	C

Matches in an alignment of two strings (ATGT) form their common subsequence

Longest common subsequence

Input: Two strings.

Output: A longest common subsequence of these strings.

Maximizing the length of a common subsequence corresponds to maximizing the score of an alignment with $\mu = \sigma = 0$.

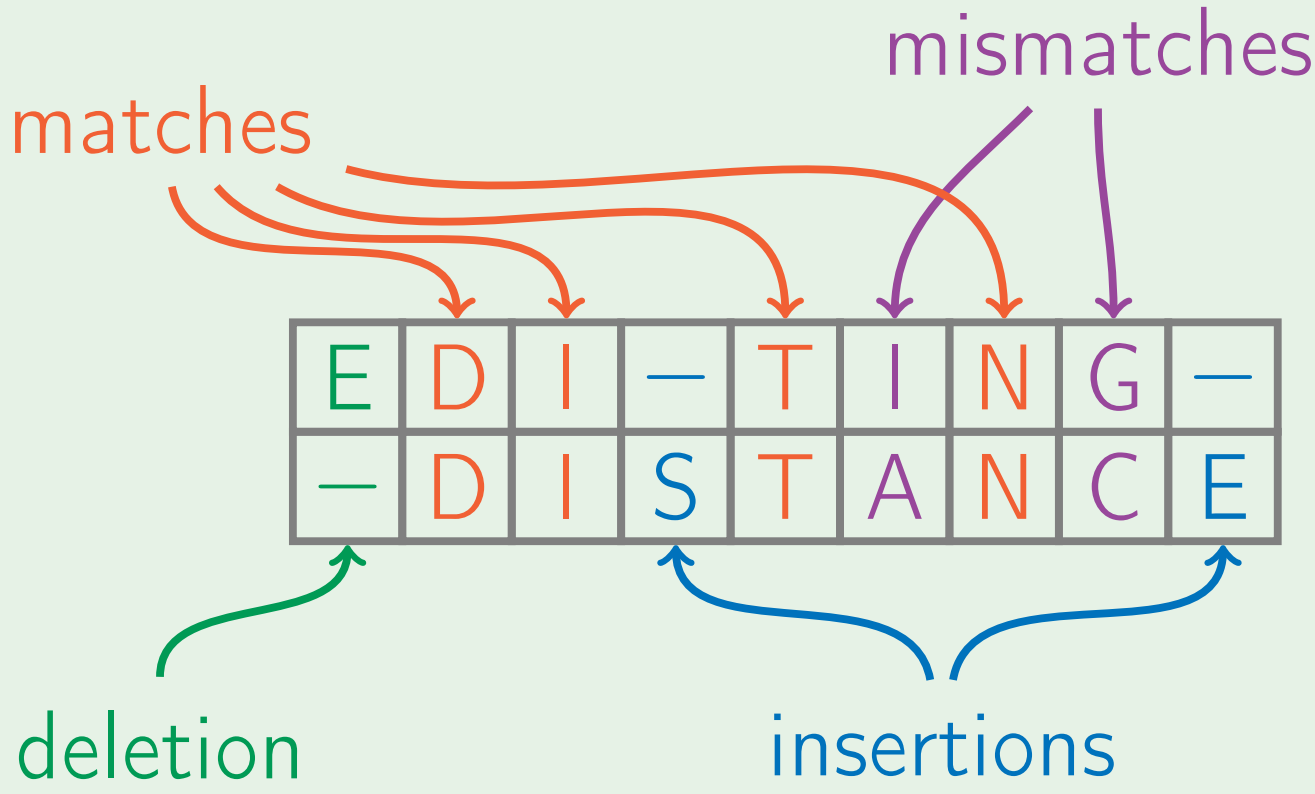
Edit distance

Input: Two strings.

Output: The minimum number of operations (insertions, deletions, and substitutions of symbols) to transform one string into another.

The minimum number of insertions, deletions and mismatches in an alignment of two strings (among all possible alignments).

Example


mismatches
matches
E D I – T I N G –
– D I S T A N C E
deletion
insertions

E	D	I	−	T	I	N	G	−
−	D	I	S	T	A	N	C	E

the total number of symbols in two strings=

$$
\begin{aligned}
&+2\cdot\#\text{matches} \\
&+2\cdot\#\text{mismatches} \\
&+1\cdot\#\text{insertions} \\
&+1\cdot\#\text{deletions}
\end{aligned}
=
\begin{aligned}
&\left.\begin{aligned}
&+2\cdot\#\text{matches} \\
&-1\cdot\#\text{insertions} \\
&-1\cdot\#\text{deletions}
\end{aligned}\right] 2\cdot\text{AlignmentScore}\ (\mu=0,\ \sigma=1/2) \\
&\qquad\qquad + \\
&\left.\begin{aligned}
&+2\cdot\#\text{mismatches} \\
&+2\cdot\#\text{insertions} \\
&+2\cdot\#\text{deletions}
\end{aligned}\right] 2\cdot\text{EditDistance}
\end{aligned}
$$

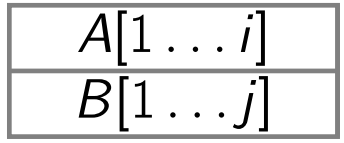
E	D	I	–	T	I	N	G	–
–	D	I	S	T	A	N	C	E

minimizing edit distance

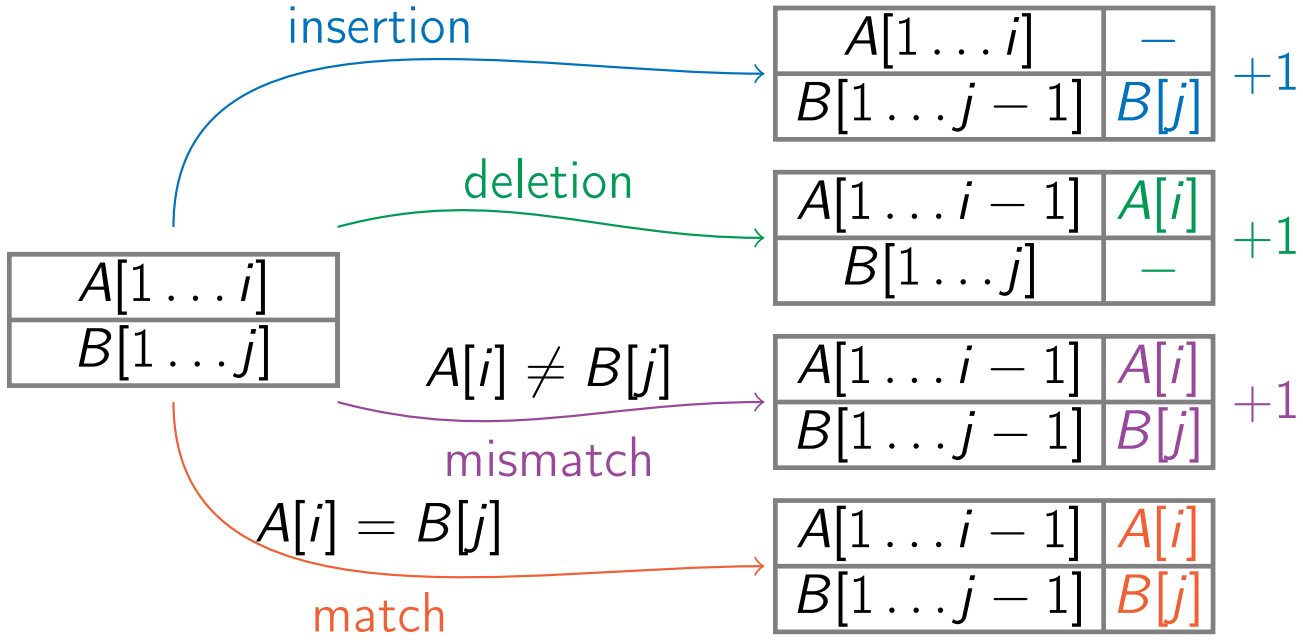
=

maximizing alignment score

Outline



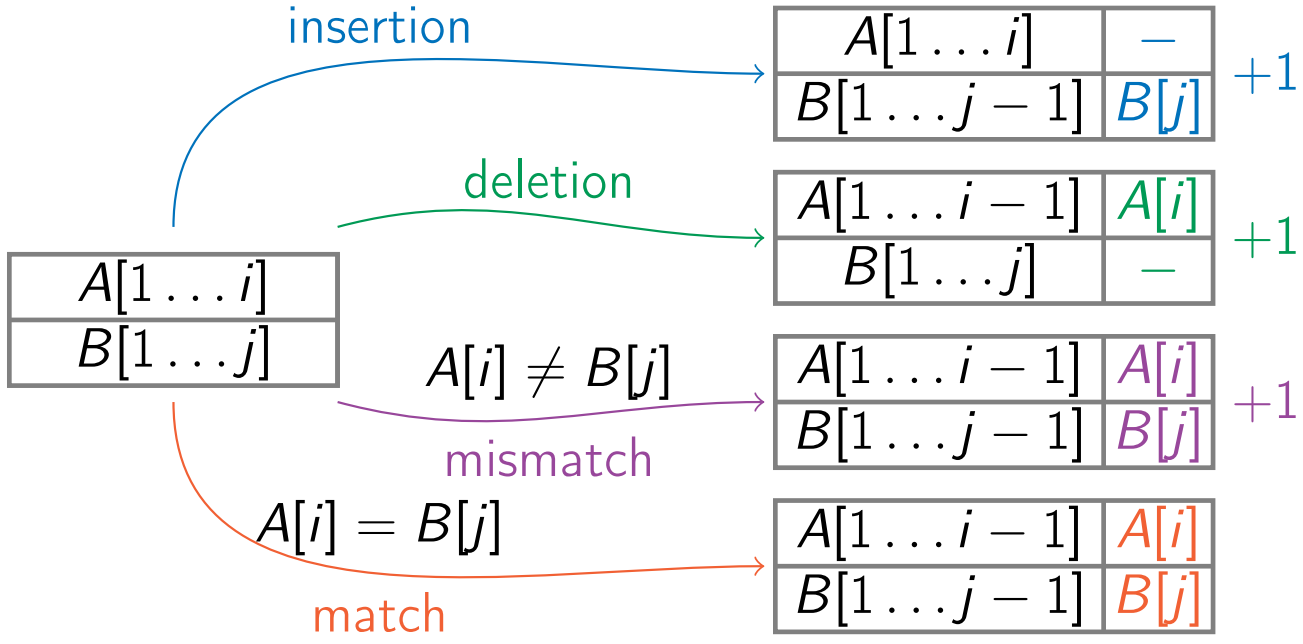
Given strings $A[1 \ldots n]$ and $B[1 \ldots m]$, what is an optimal alignment (an alignment that results in minimum edit distance) of an i-prefix $A[1 \ldots i]$ of the first string and a j-prefix $B[1 \ldots j]$ of the second string?



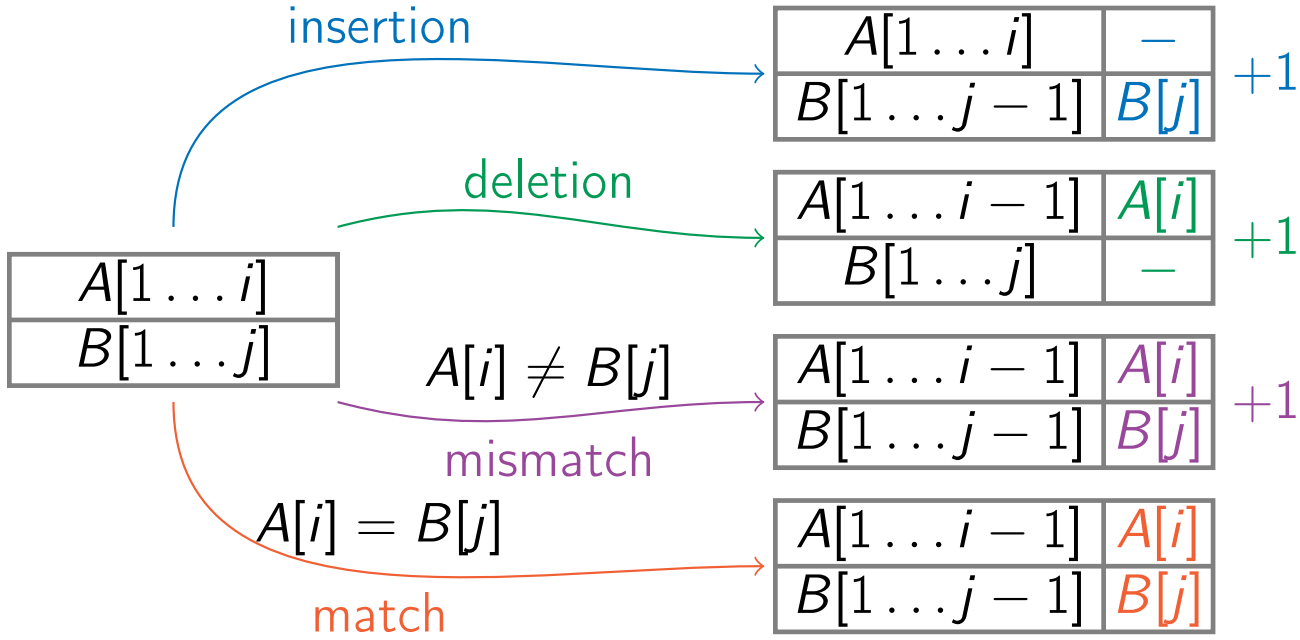
The last column of an optimal alignment is either

- an insertion,
- a deletion,
- a mismatch,
- or a match.

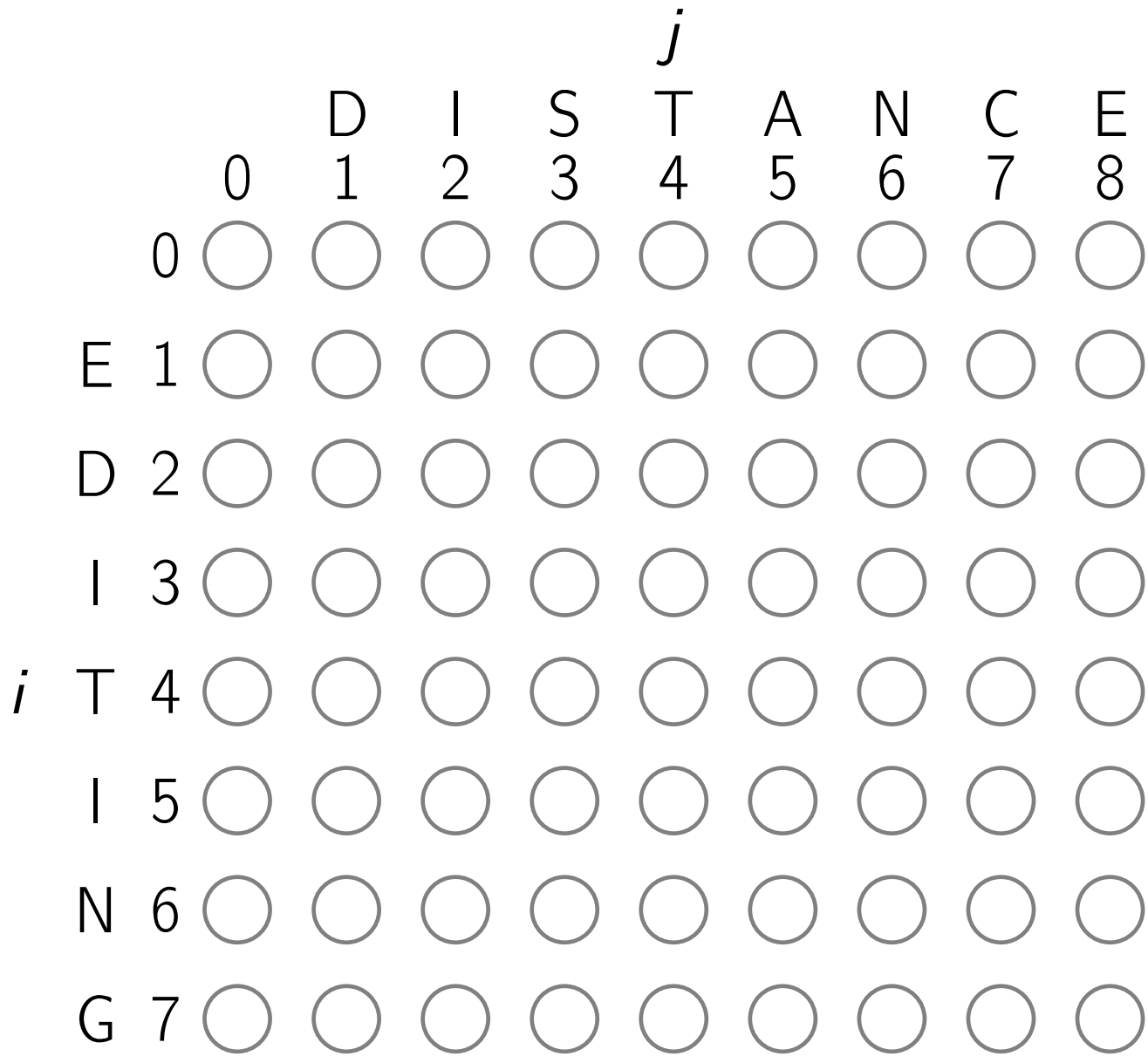
What is left (after the removal of the last column) is an **optimal** alignment of the corresponding two prefixes.



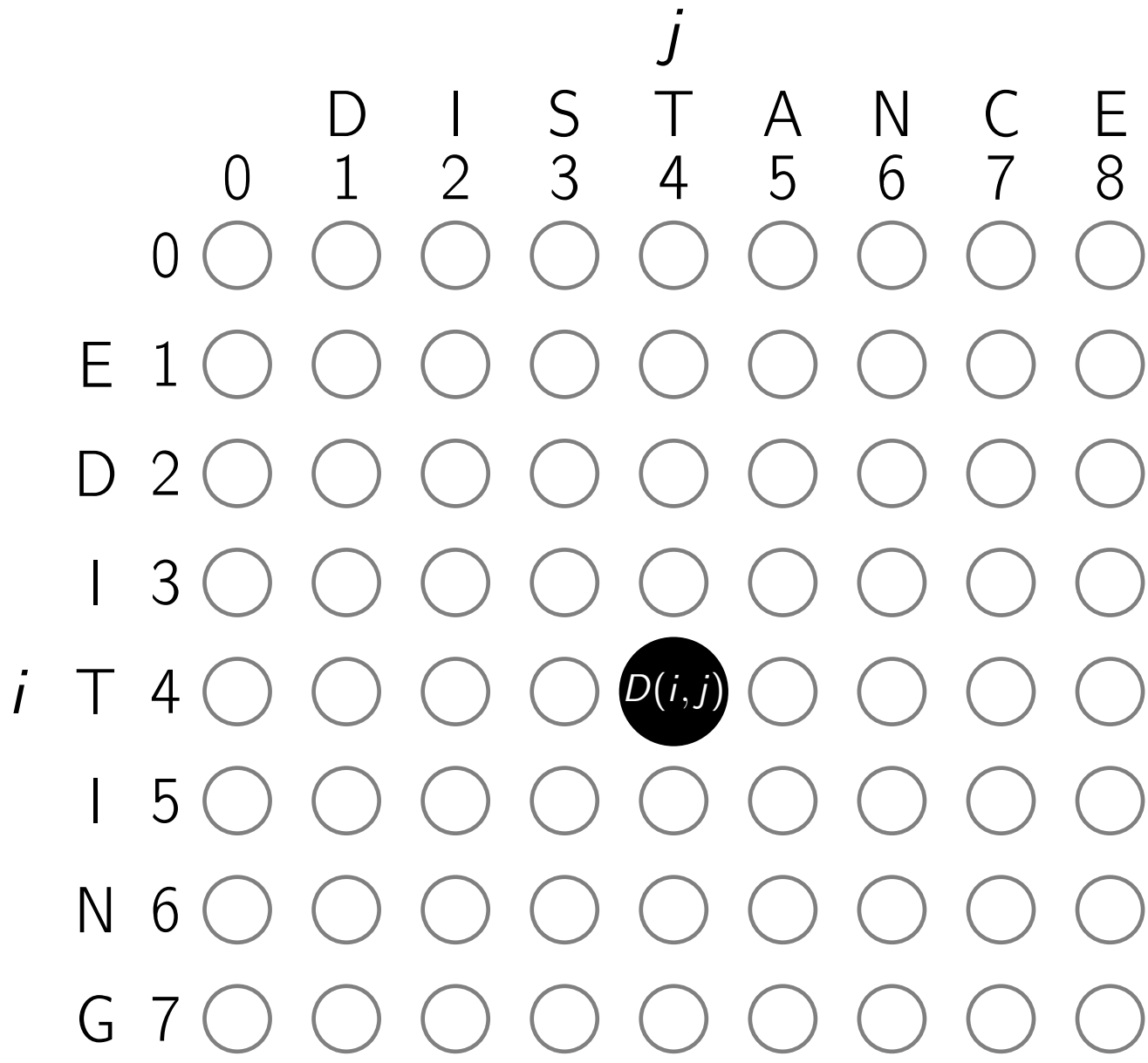
Let $D(i, j)$ be the edit distance of an i-prefix $A[1 \ldots i]$ and a j-prefix $B[1 \ldots j]$.



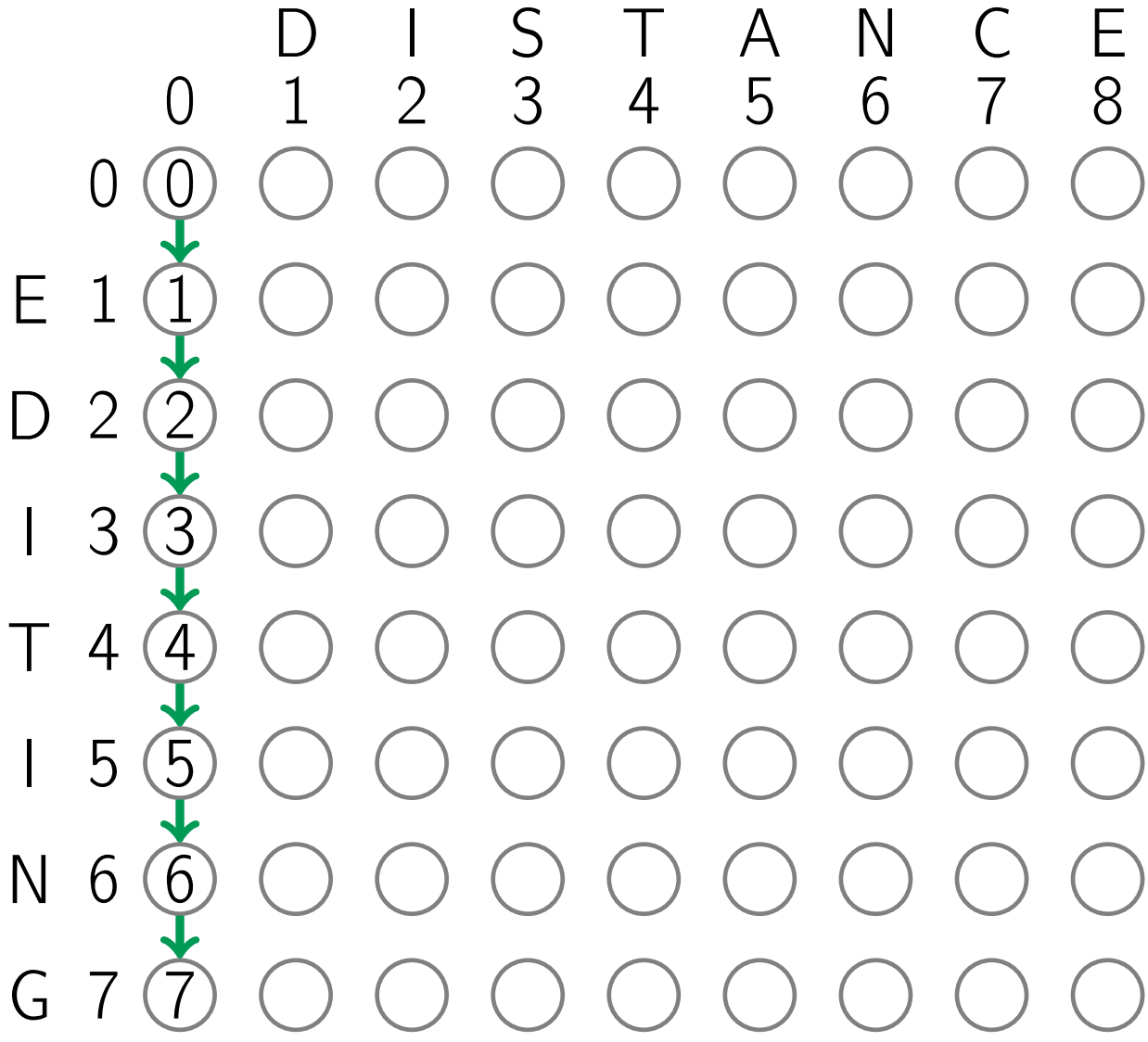
$$D(i,j) = \min \begin{cases} D(i,j-1)+1 & \\ D(i-1,j)+1 & \\ D(i-1,j-1)+1 & \text{if } A[i] \neq B[j] \\ D(i-1,j-1) & \text{if } A[i] = B[j] \end{cases}$$

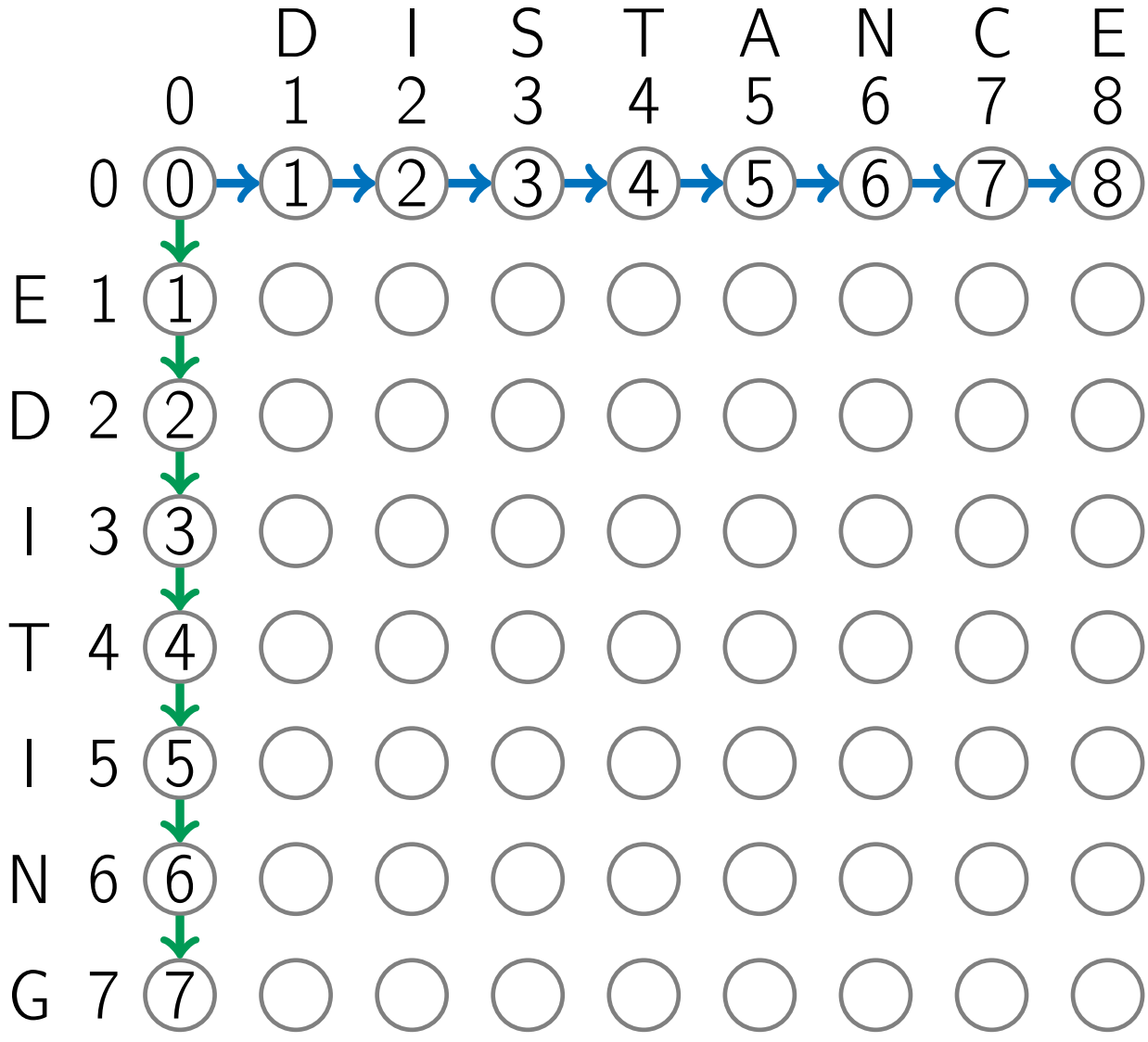


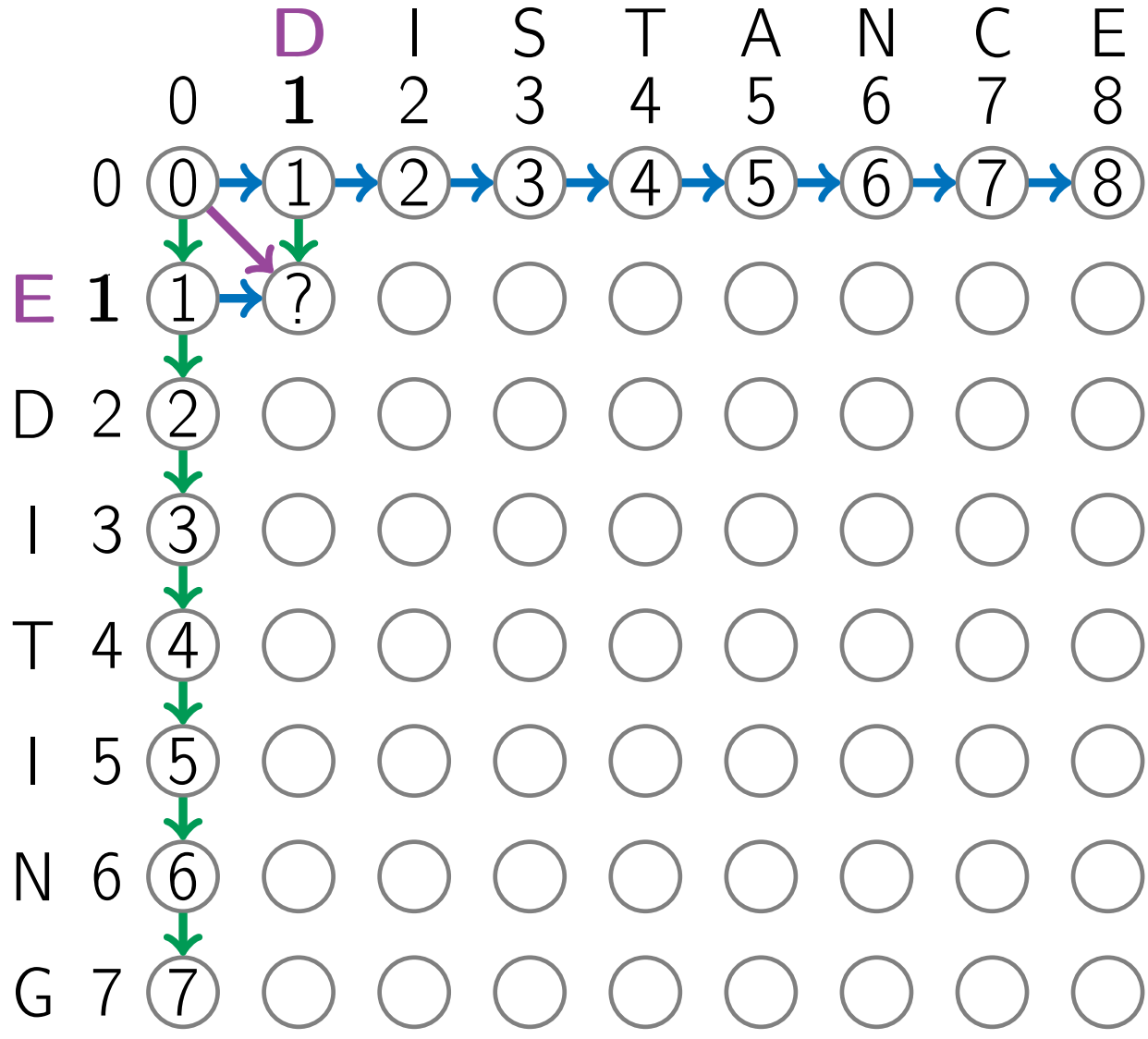
comparing $A[1 \ldots n] =$ EDITING
and $B[1 \ldots m] =$ DISTANCE



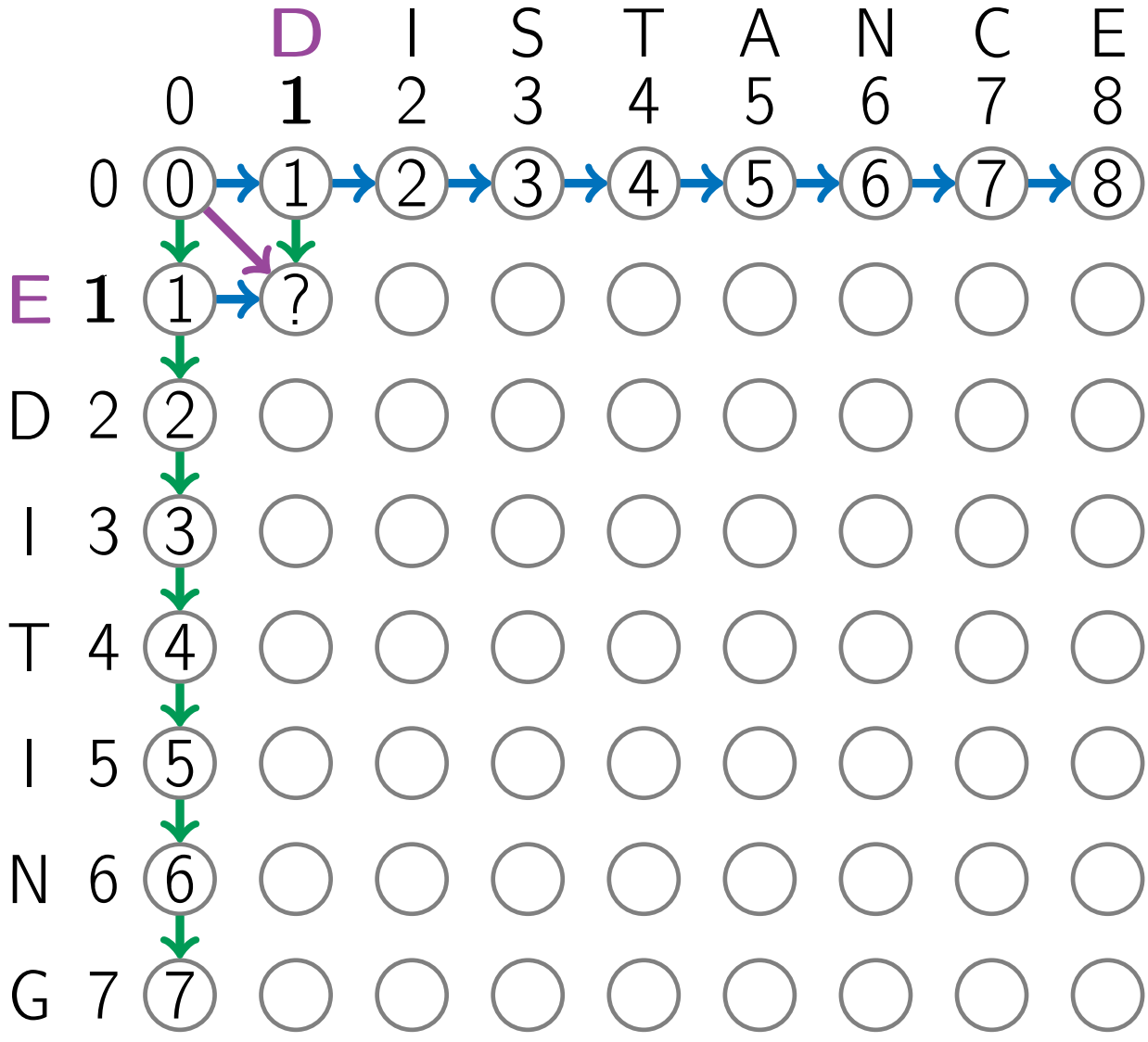
comparing $A[1 \ldots n] =$ EDITING
and $B[1 \ldots m] =$ DISTANCE


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0
E 1 1
D 2 2
I 3 3
T 4 4
I 5 5
N 6 6
G 7 7

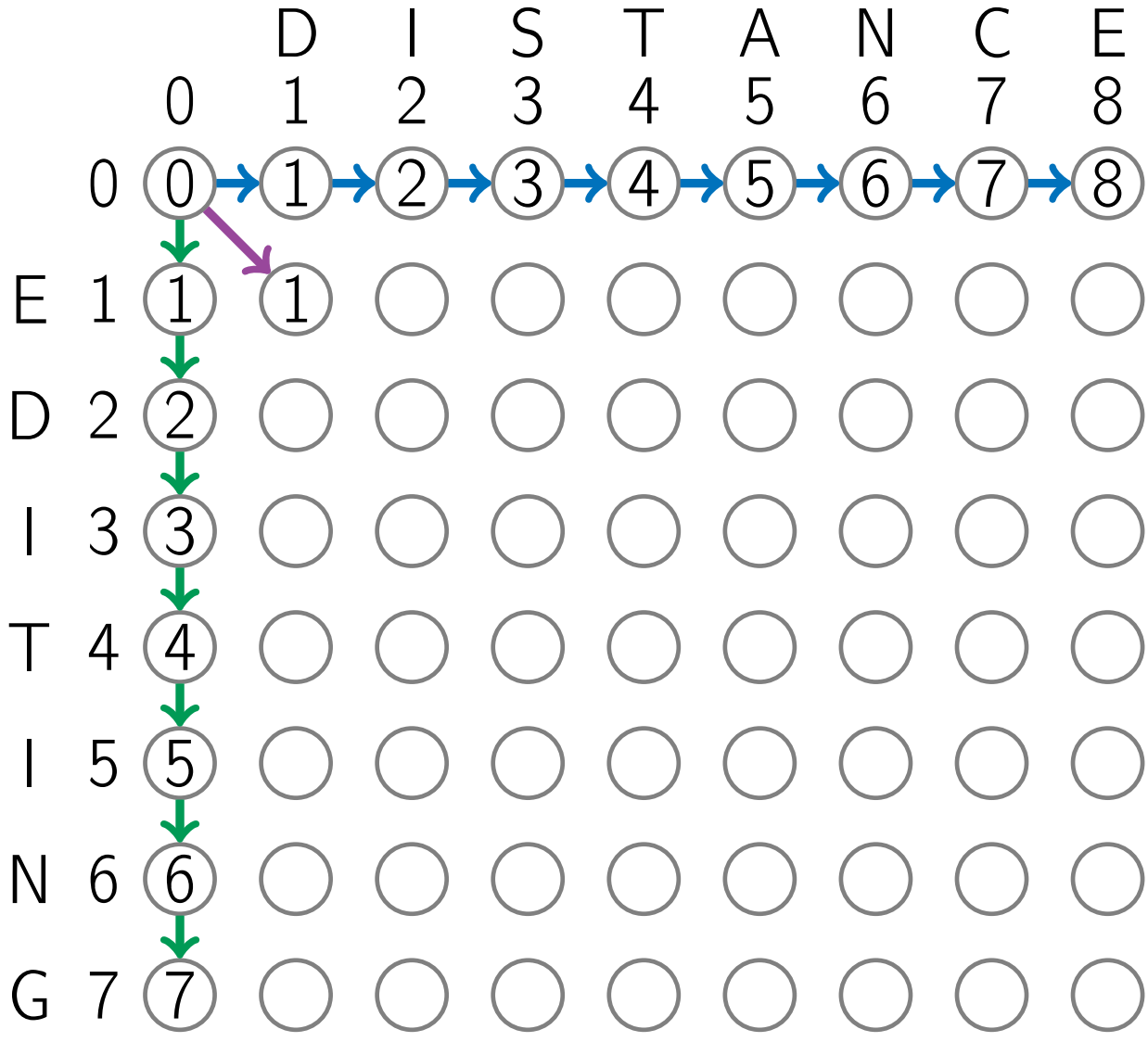

D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1
D 2 2
I 3 3
T 4 4
I 5 5
N 6 6
G 7 7

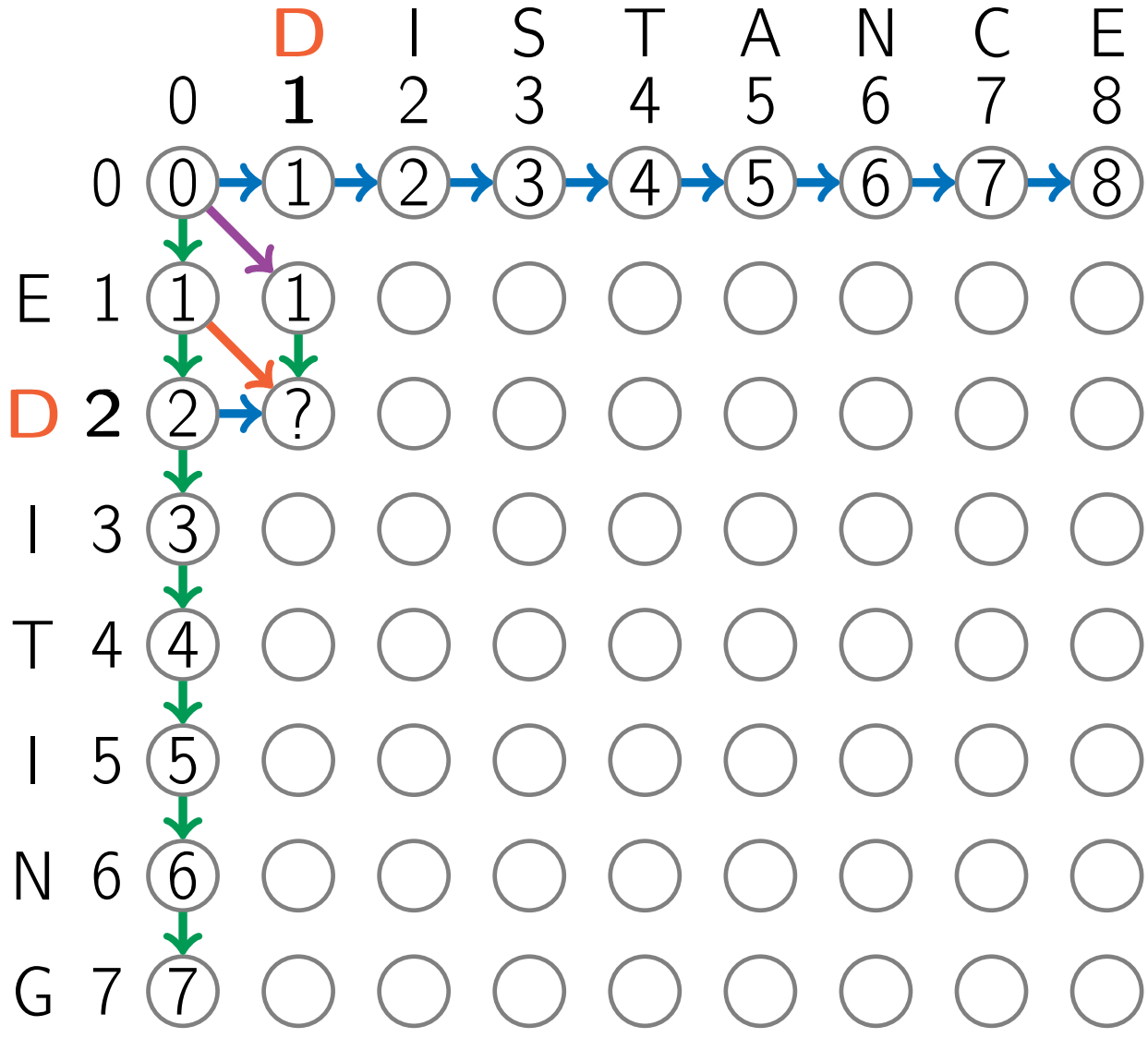


$$D(1,1) = \min\{D(1,0)+1, D(0,1)+1, D(0,0)+1\}$$

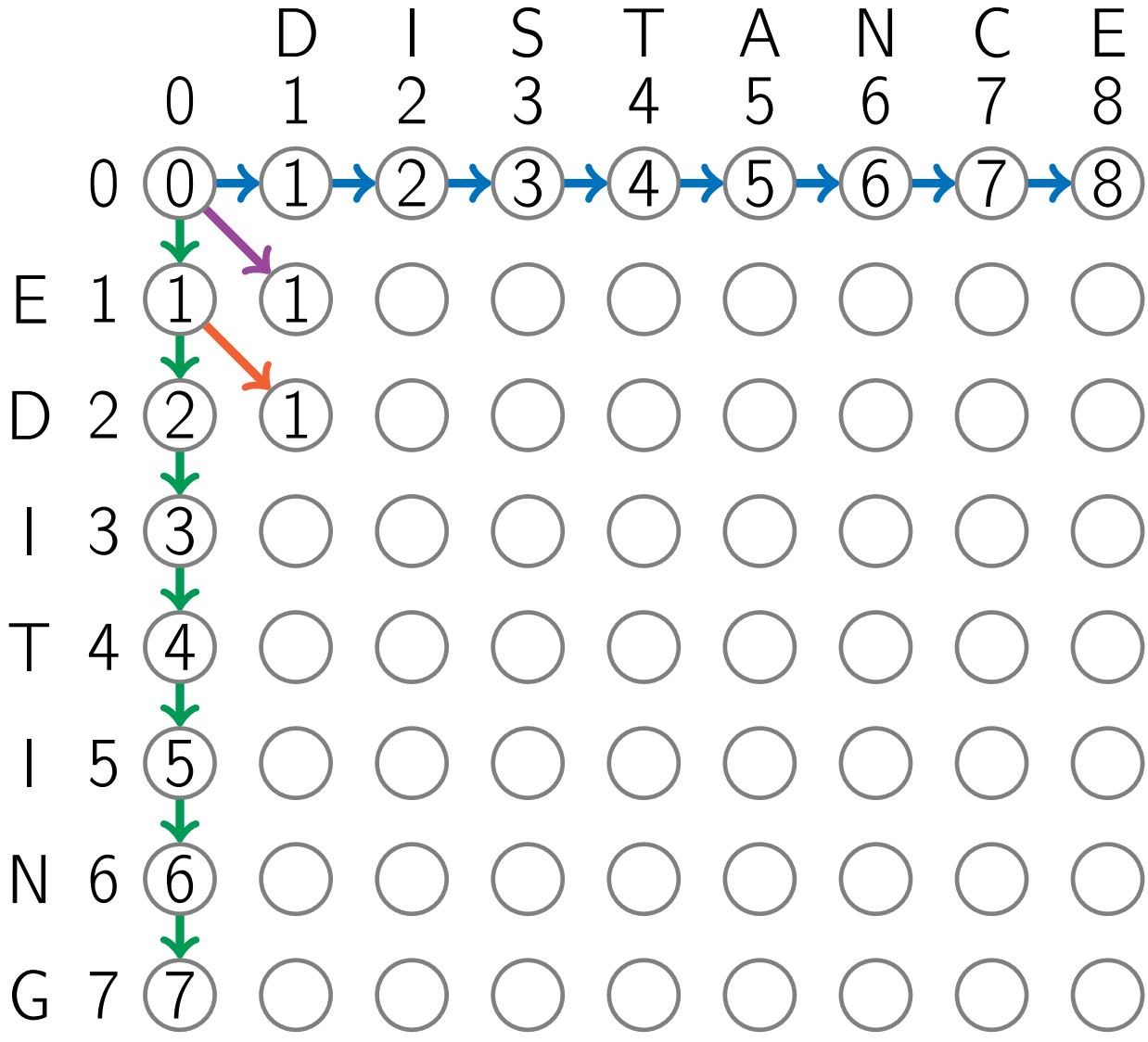


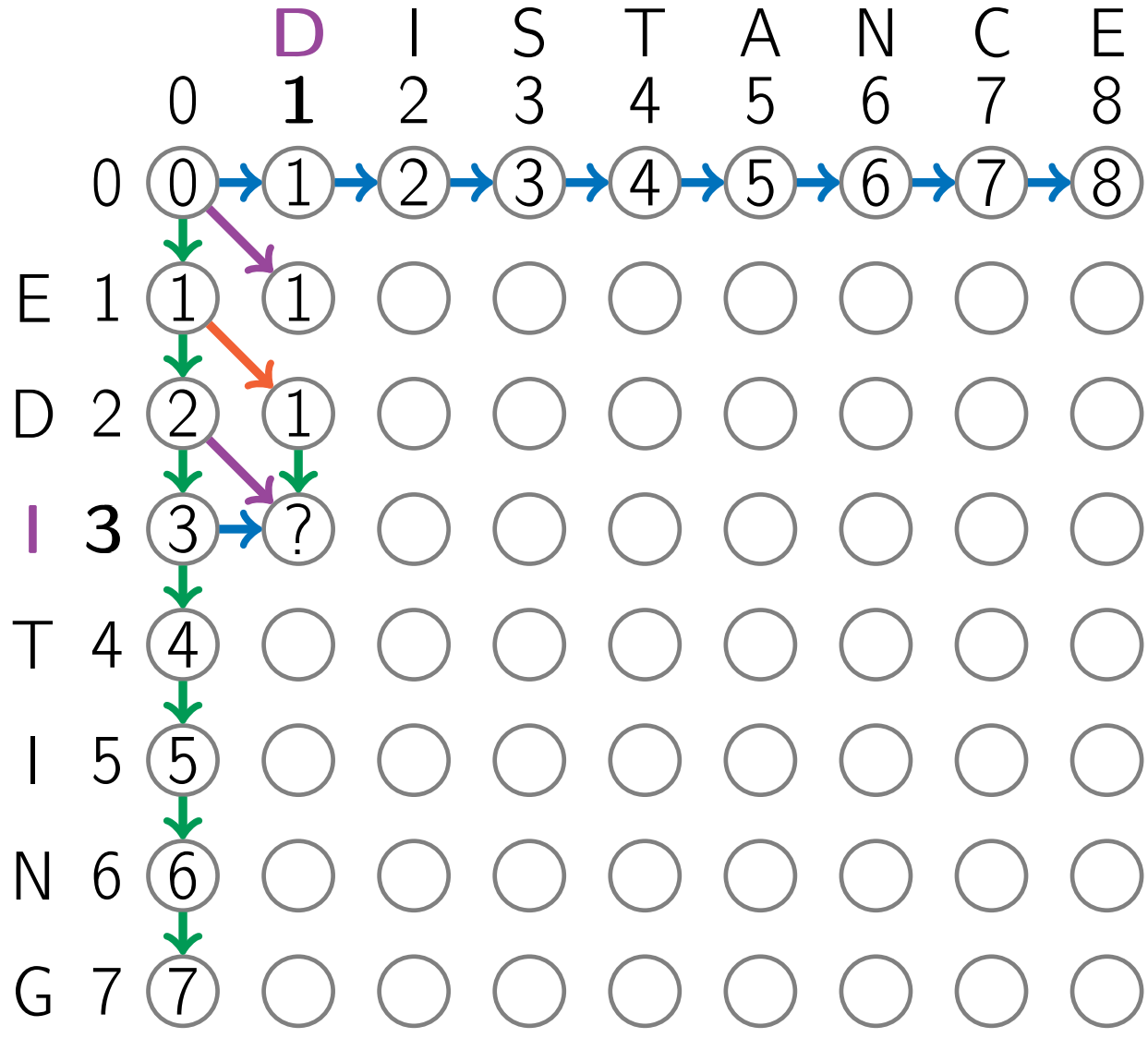
$$D(1,1) = \min\{2, 2, 1\}$$


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1
D 2 2
I 3 3
T 4 4
I 5 5
N 6 6
G 7 7

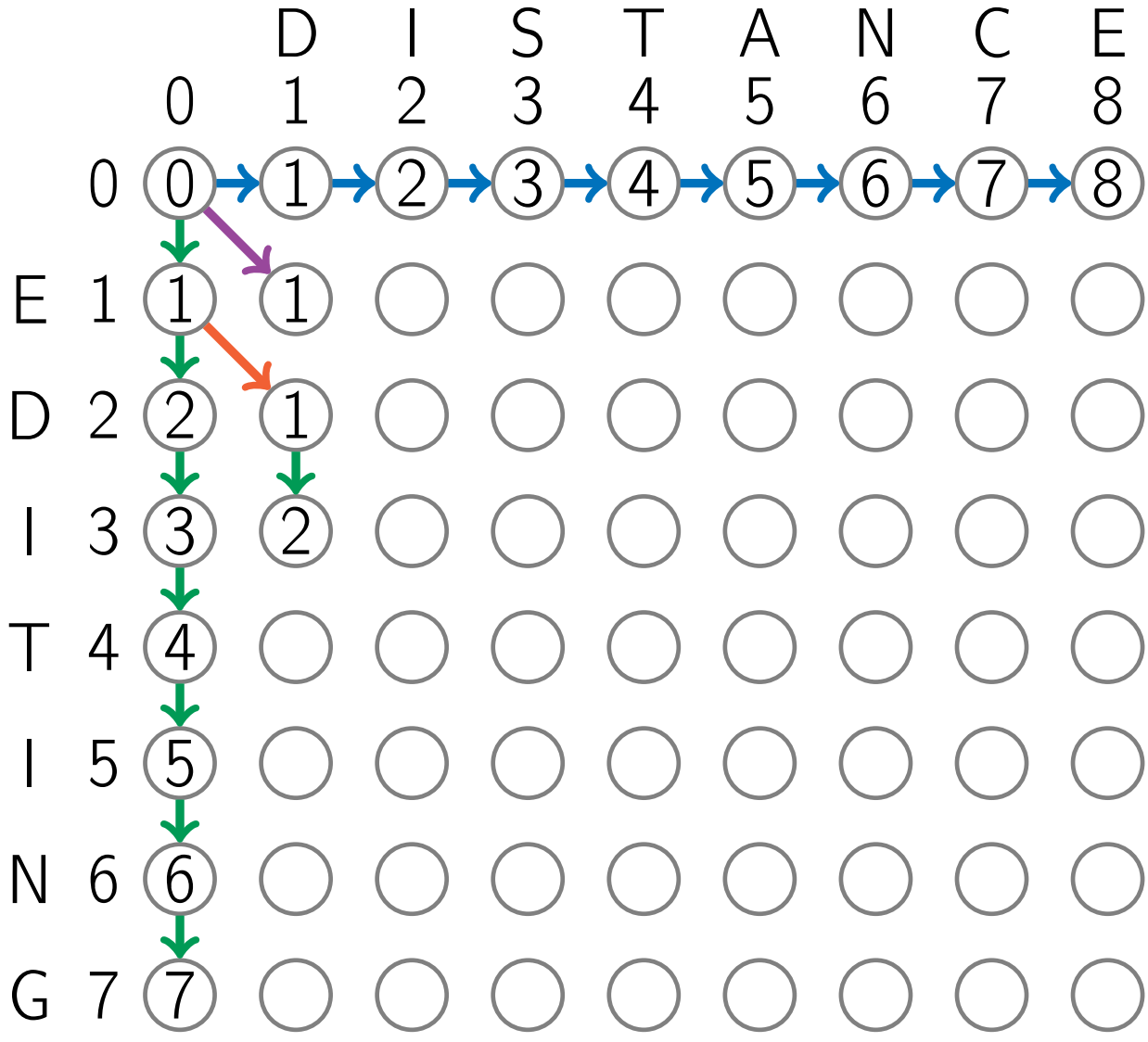


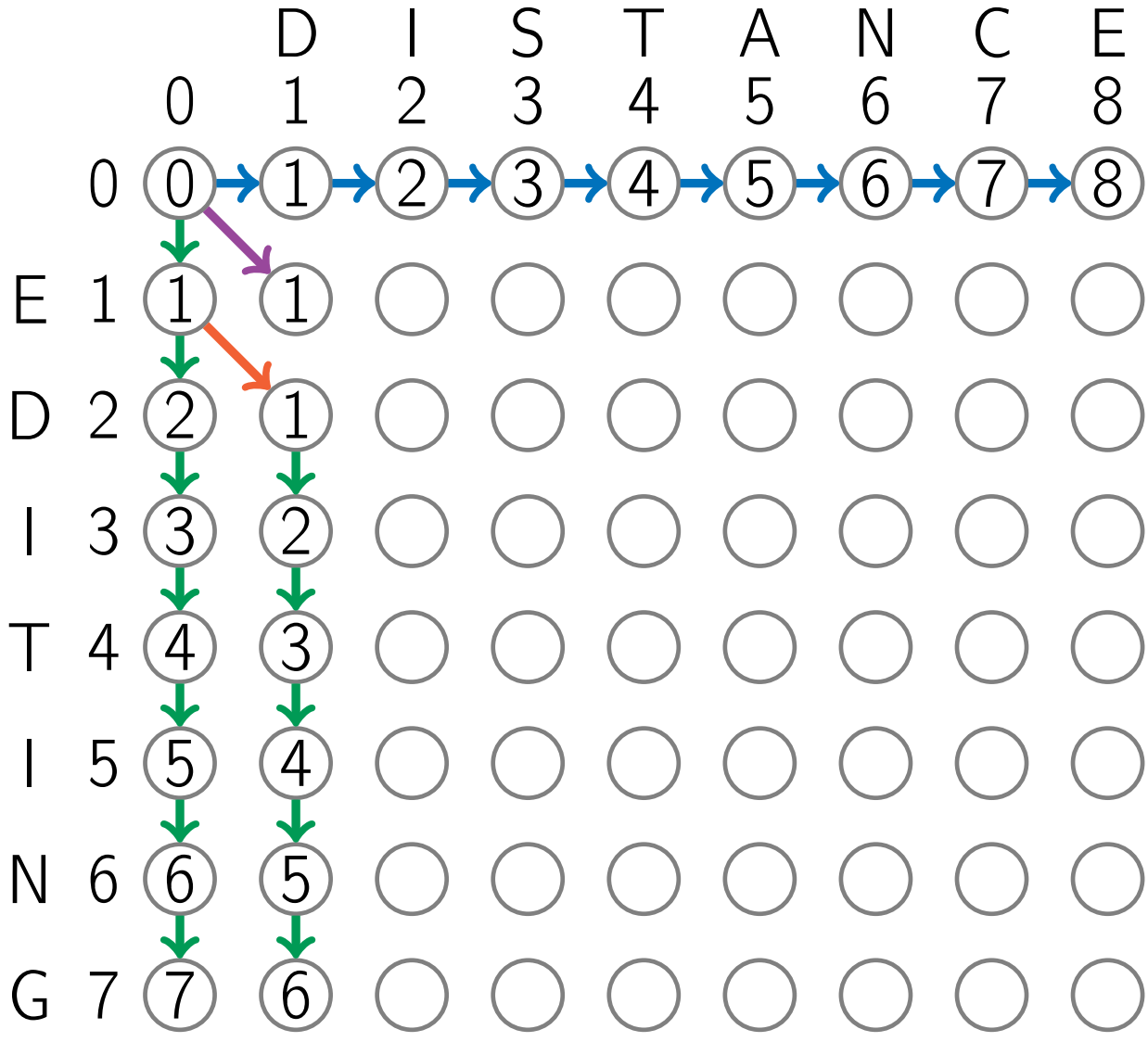
$$D(2,1) = \min\{D(2,0)+1, D(1,1)+1, D(1,0)\}$$

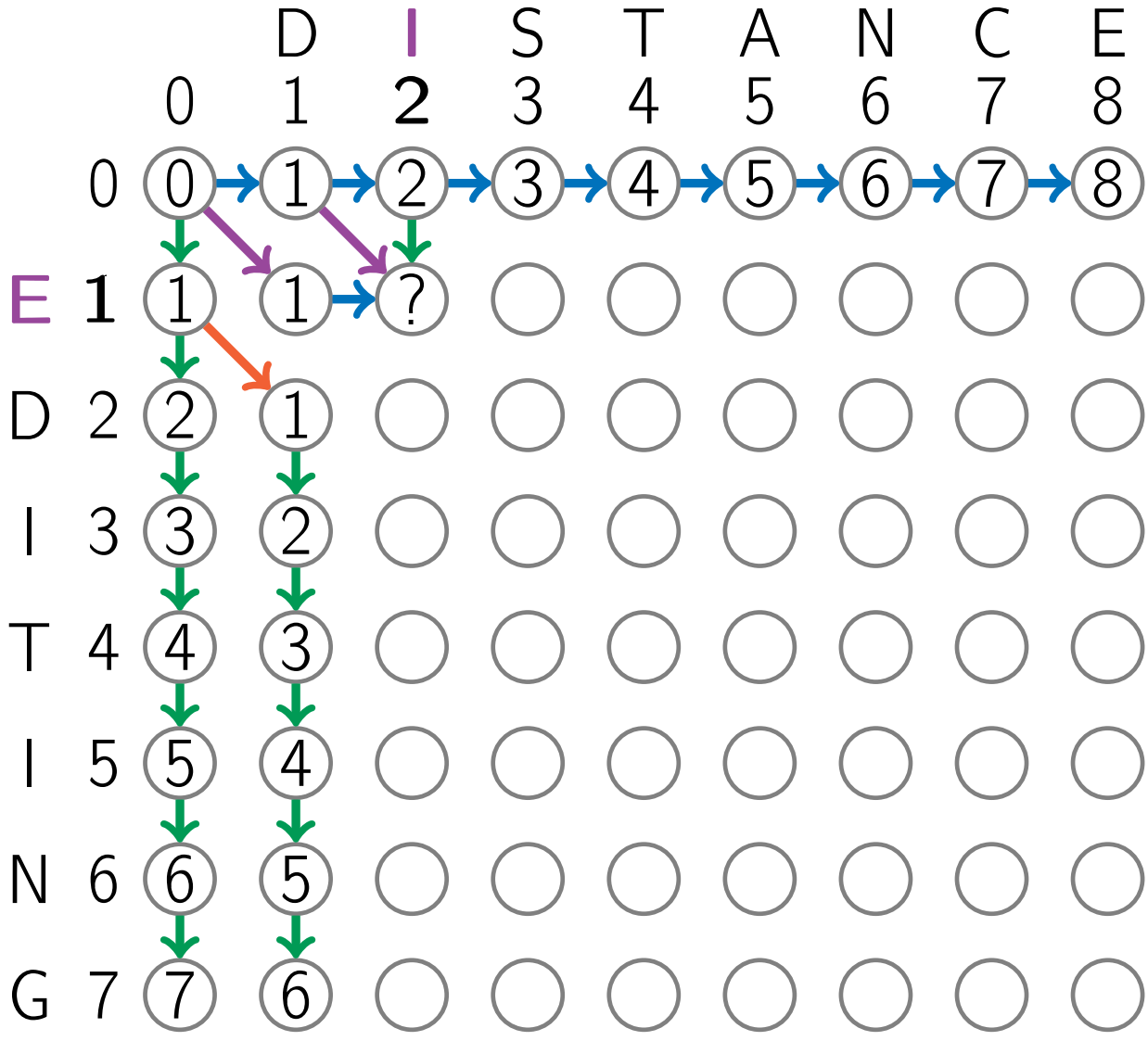

D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1
D 2 2 1
I 3 3
T 4 4
I 5 5
N 6 6
G 7 7



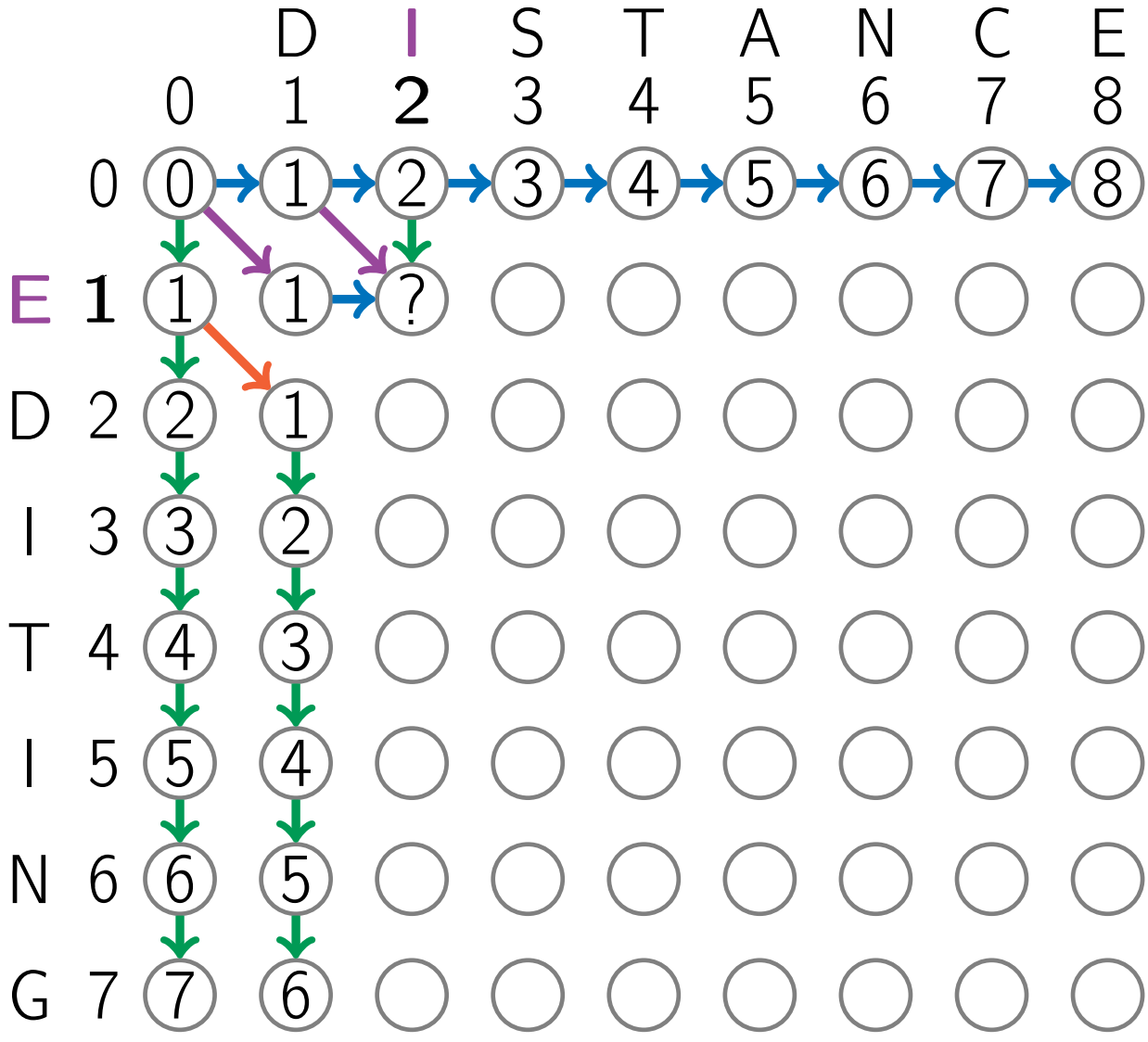
$$D(3,1) = \min\{D(3,0)+1, D(2,1)+1, D(2,0)+1\}$$


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1
D 2 2 1
I 3 3 2
T 4 4
I 5 5
N 6 6
G 7 7

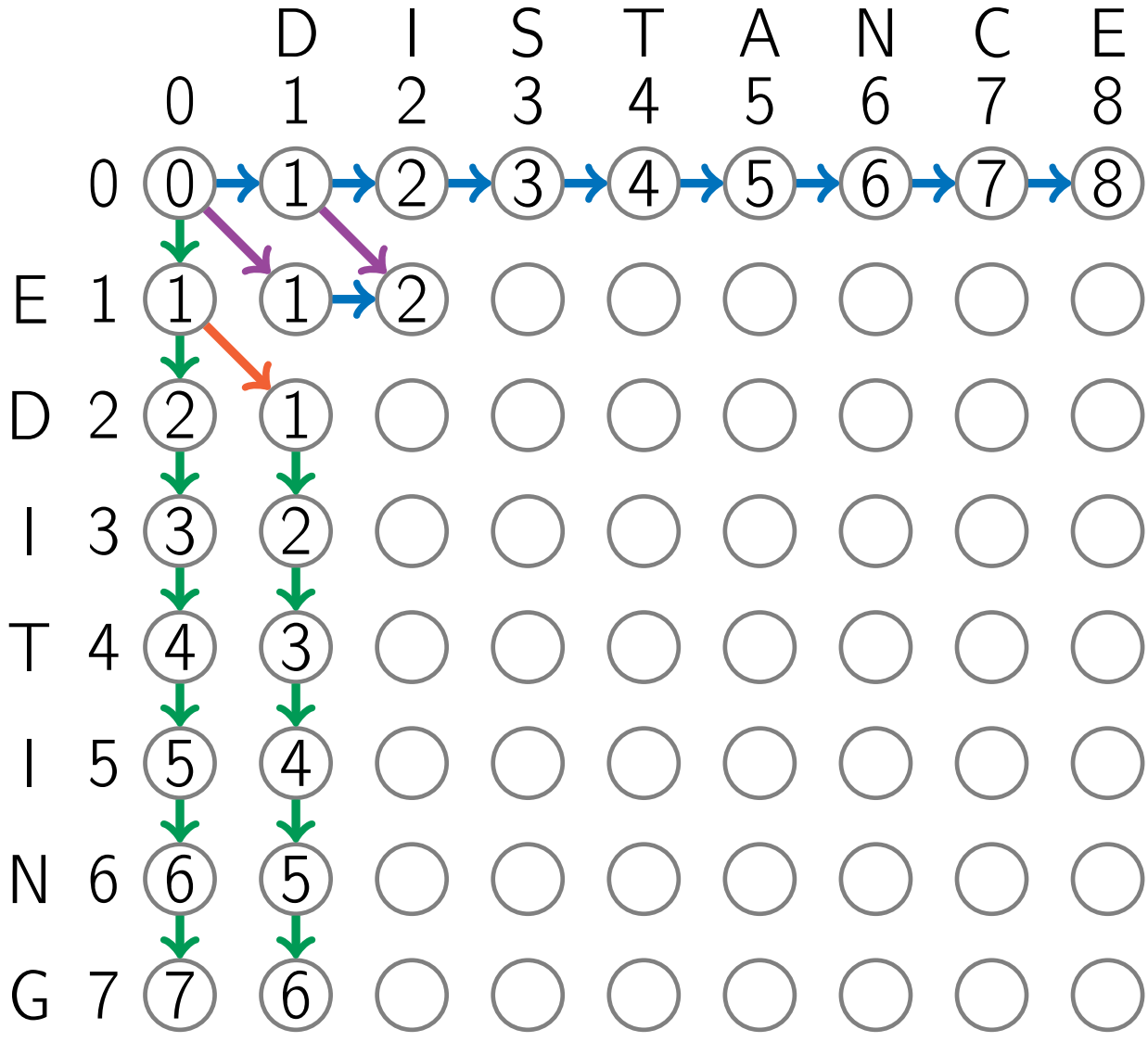

D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1
D 2 2 1
I 3 3 2
T 4 4 3
I 5 5 4
N 6 6 5
G 7 7 6

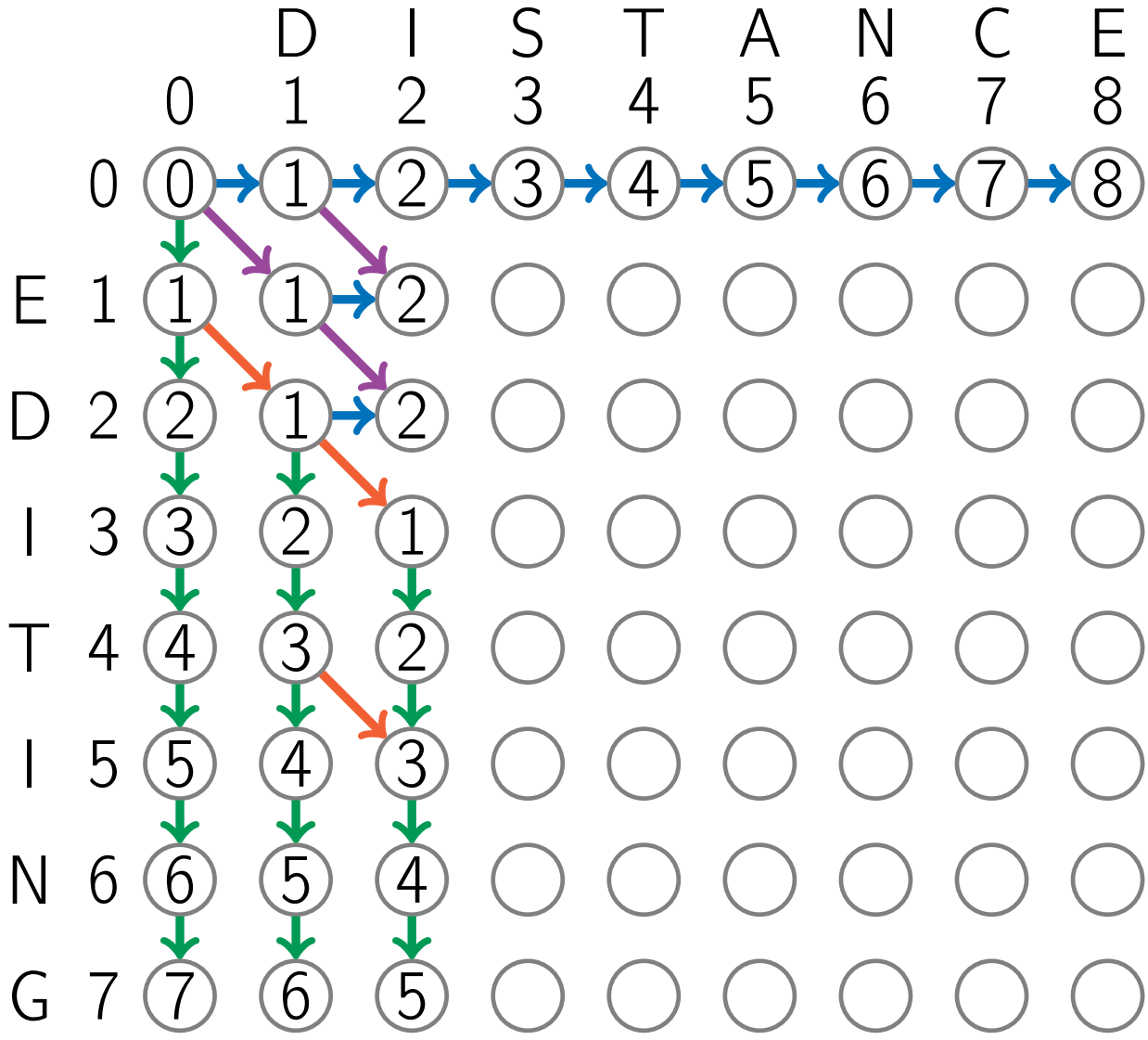


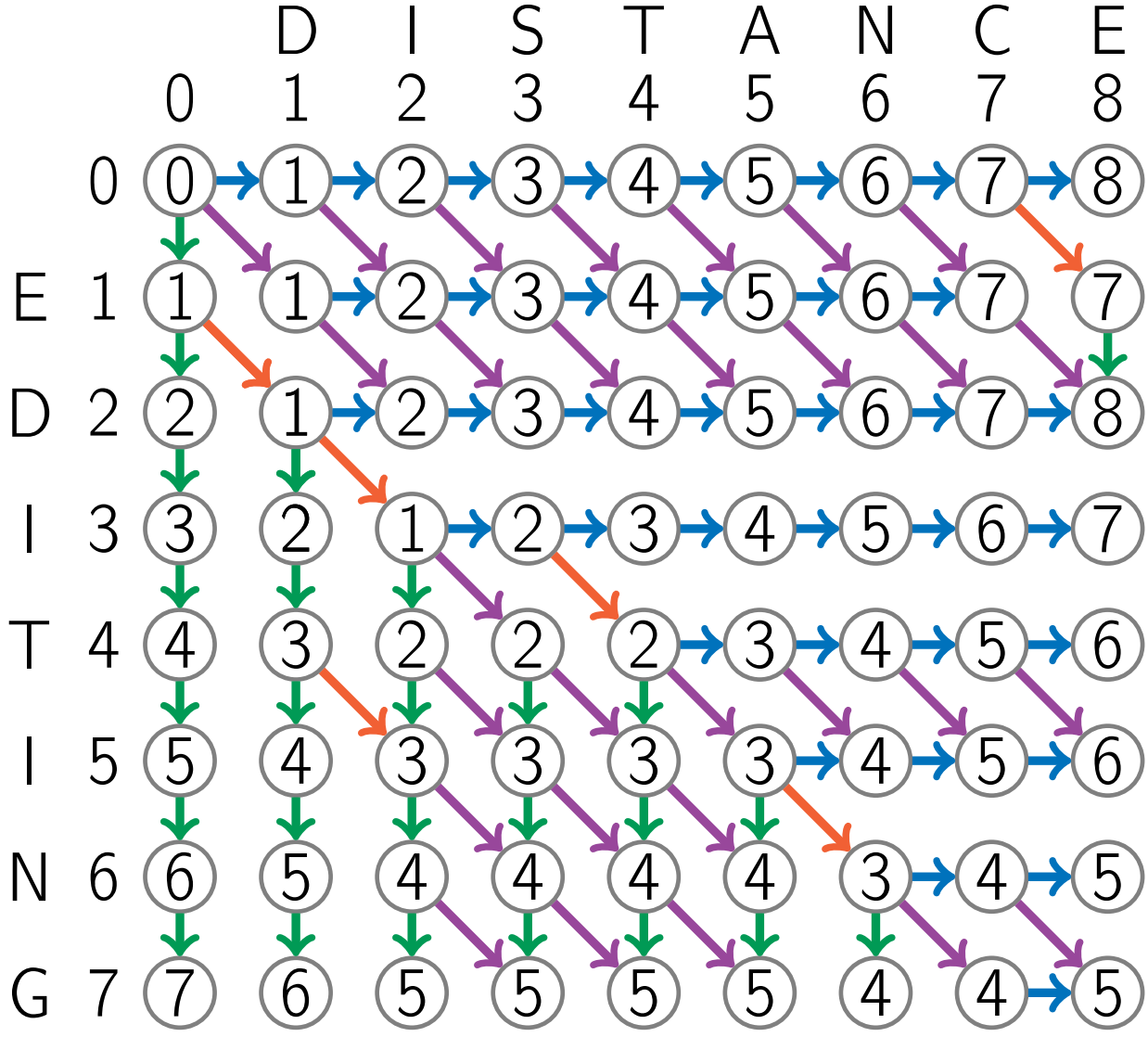
$$D(1,2) = \min\{D(1,1)+1, D(0,2)+1, D(0,1)+1\}$$


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1 ?
D 2 2 1
I 3 3 2
T 4 4 3
I 5 5 4
N 6 6 5
G 7 7 6

$$D(1,1) = \min\{2, 3, 2\}$$


D I S T A N C E
0 1 2 3 4 5 6 7 8
0
E 1
D 2
I 3
T 4
I 5
N 6
G 7


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1 2
D 2 2 1 2
I 3 3 2 1
T 4 4 3 2
I 5 5 4 3
N 6 6 5 4
G 7 7 6 5


D I S T A N C E
0 1 2 3 4 5 6 7 8
0
E 1
D 2
I 3
T 4
I 5
N 6
G 7

EditDistance($A[1 \ldots n], B[1 \ldots m]$)

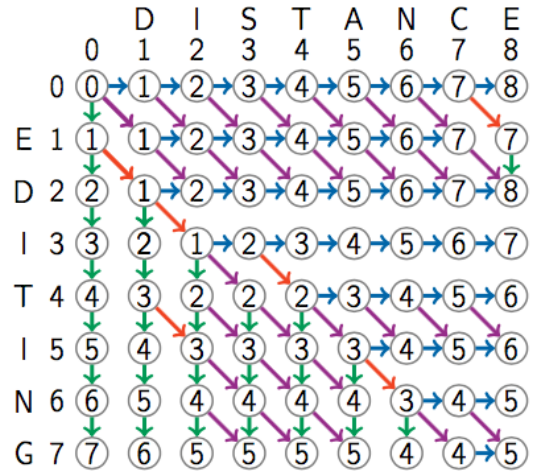
$D(i, 0) \leftarrow i$ and $D(0, j) \leftarrow j$ for all i, j

```
for j from 1 to m:
  for i from 1 to n:
    insertion ← D(i, j − 1) + 1
    deletion ← D(i − 1, j) + 1
    match ← D(i − 1, j − 1)
    mismatch ← D(i − 1, j − 1) + 1
    if A[i] = B[j]:
      D(i, j) ← min(insertion, deletion, match)
    else:
      D(i, j) ← min(insertion, deletion, mismatch)
return D(n, m)
```

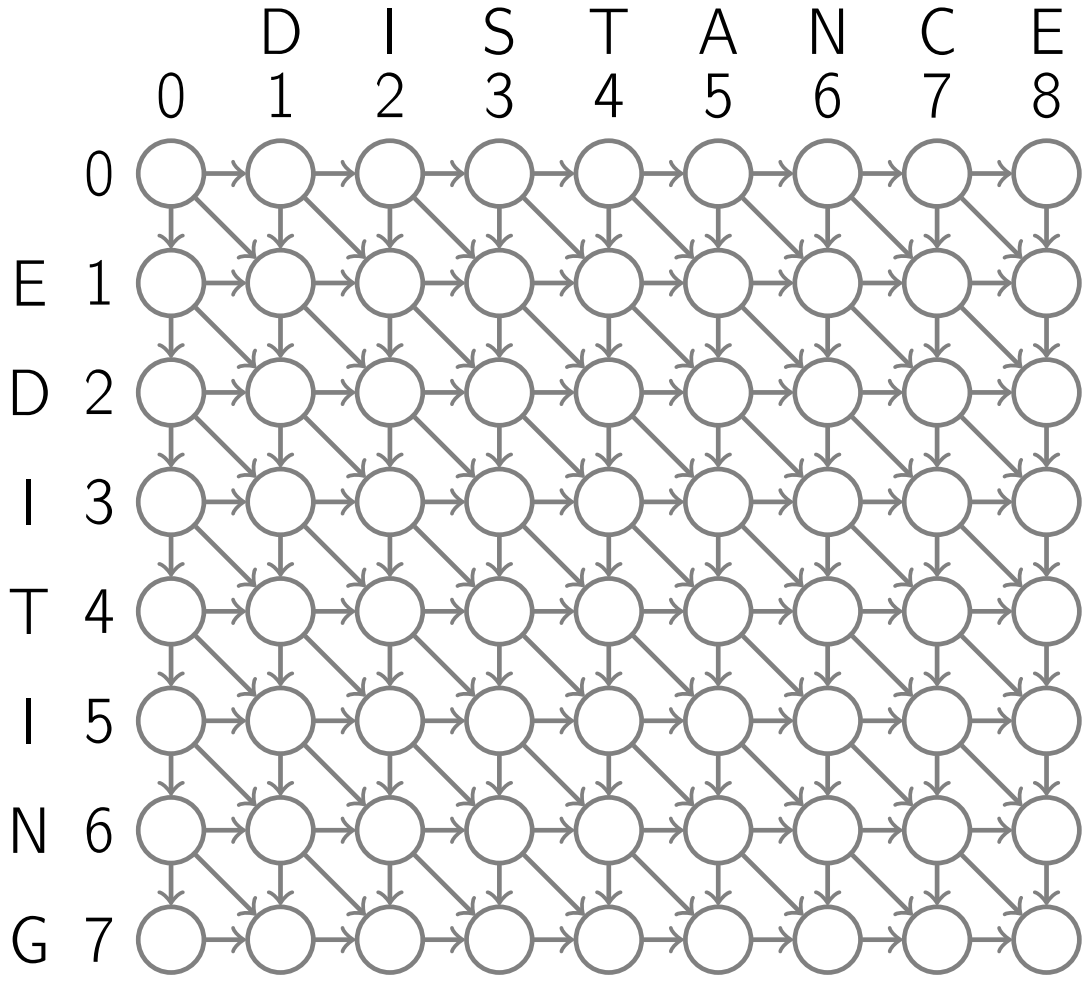
Outline

Optimal Alignment

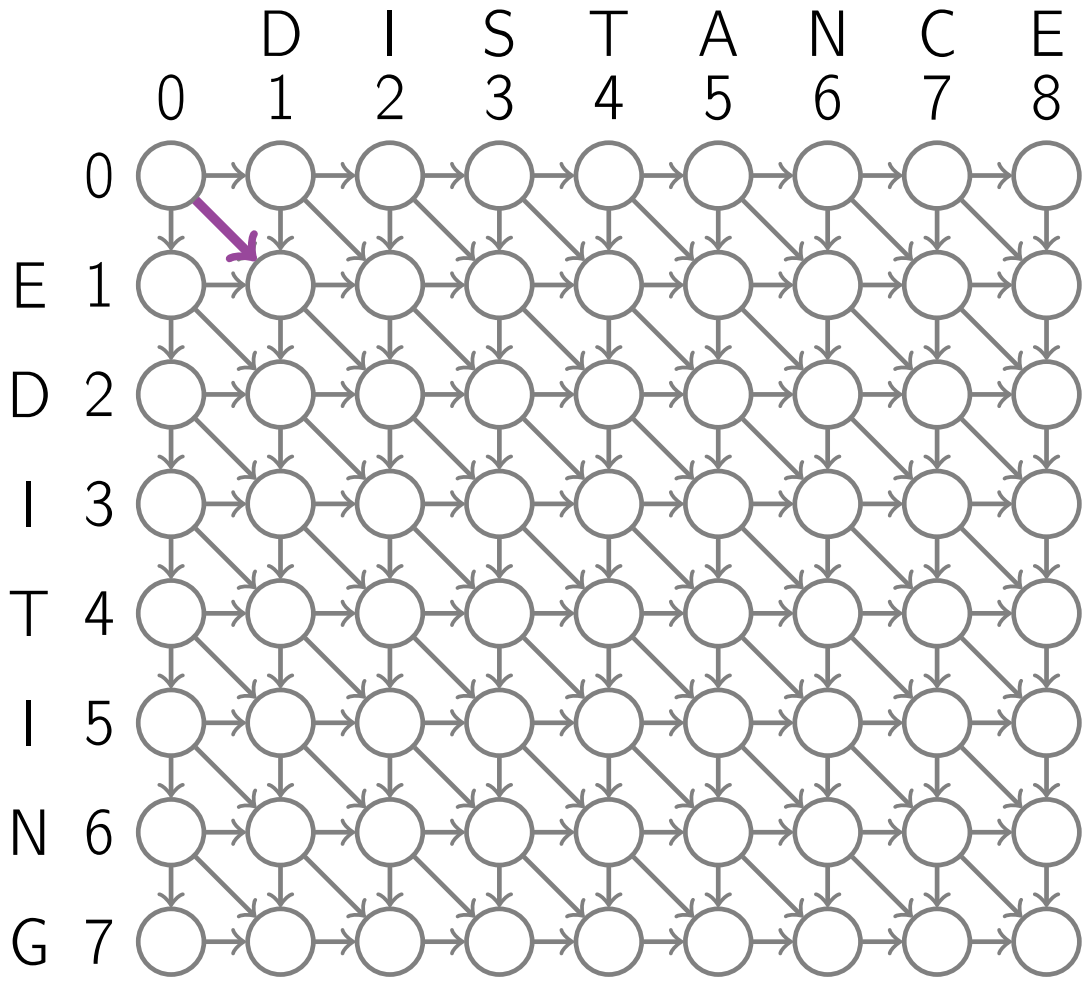
- We have computed the edit distance, but how can we find an optimal alignment?
- The backtracking pointers that we stored will help us to reconstruct an optimal alignment.



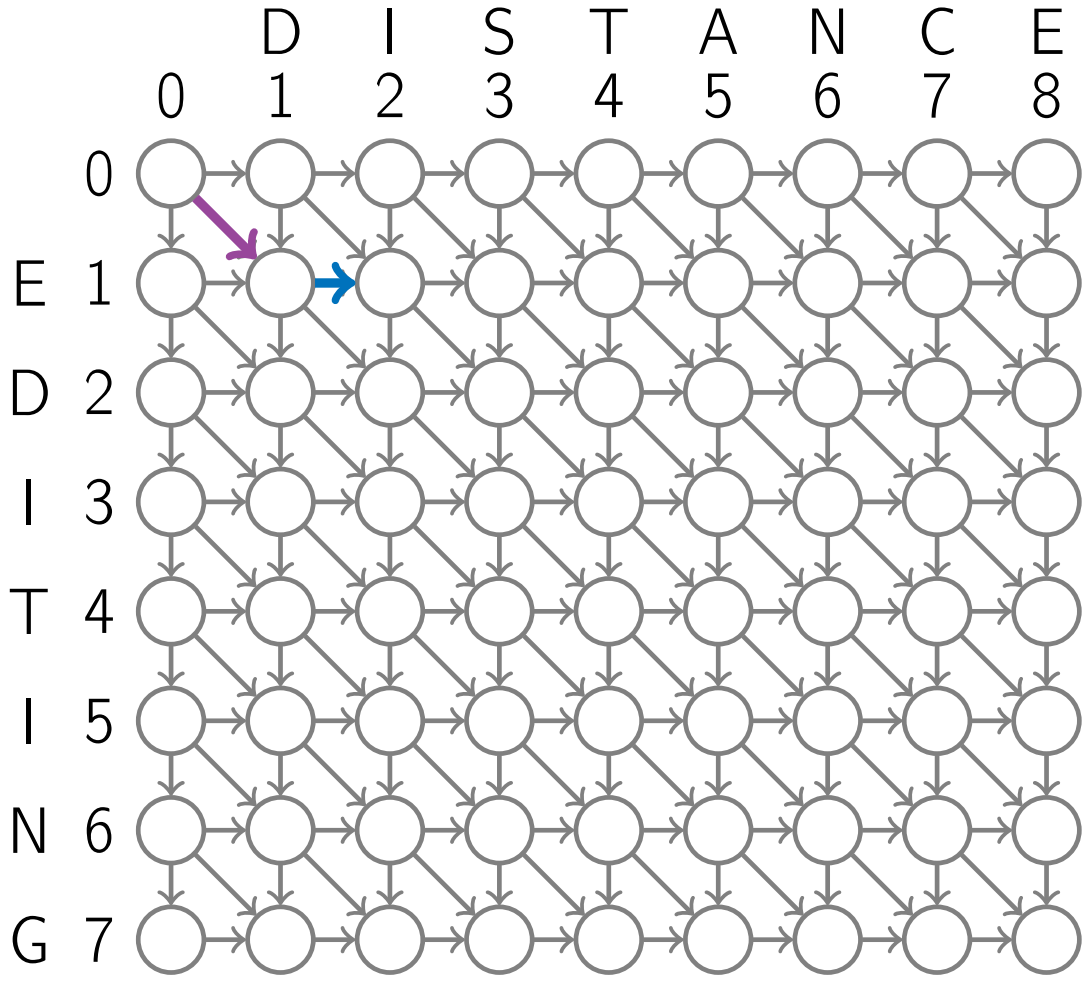
any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$

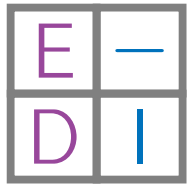


any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$

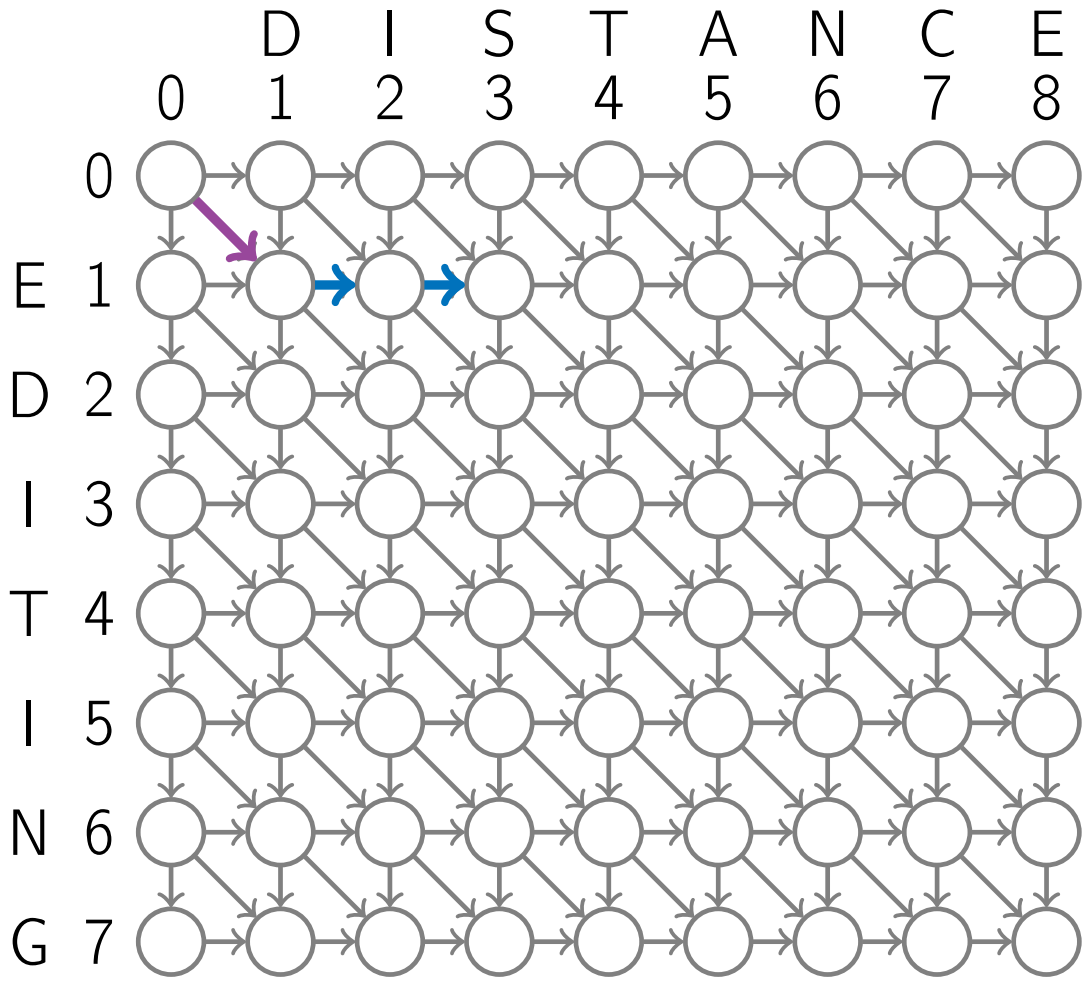


any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$



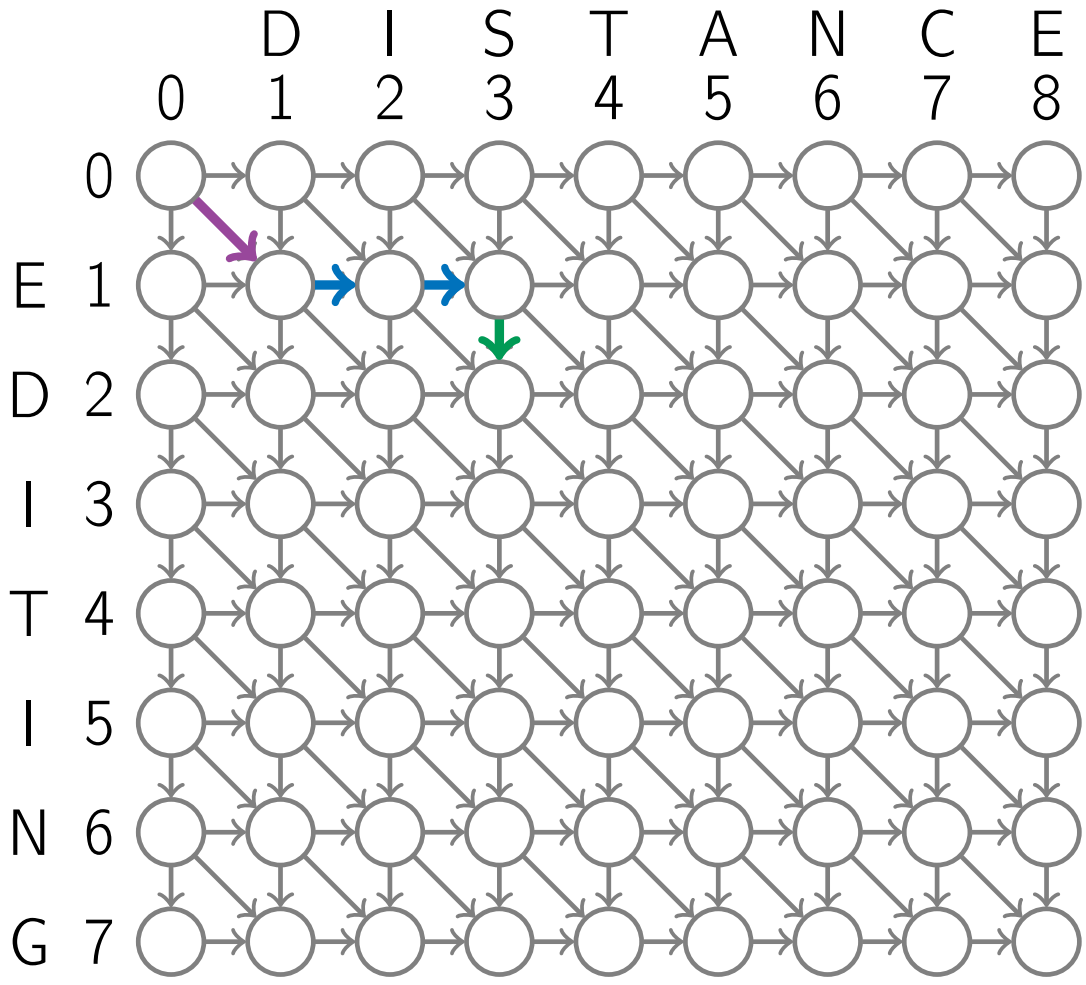


any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$



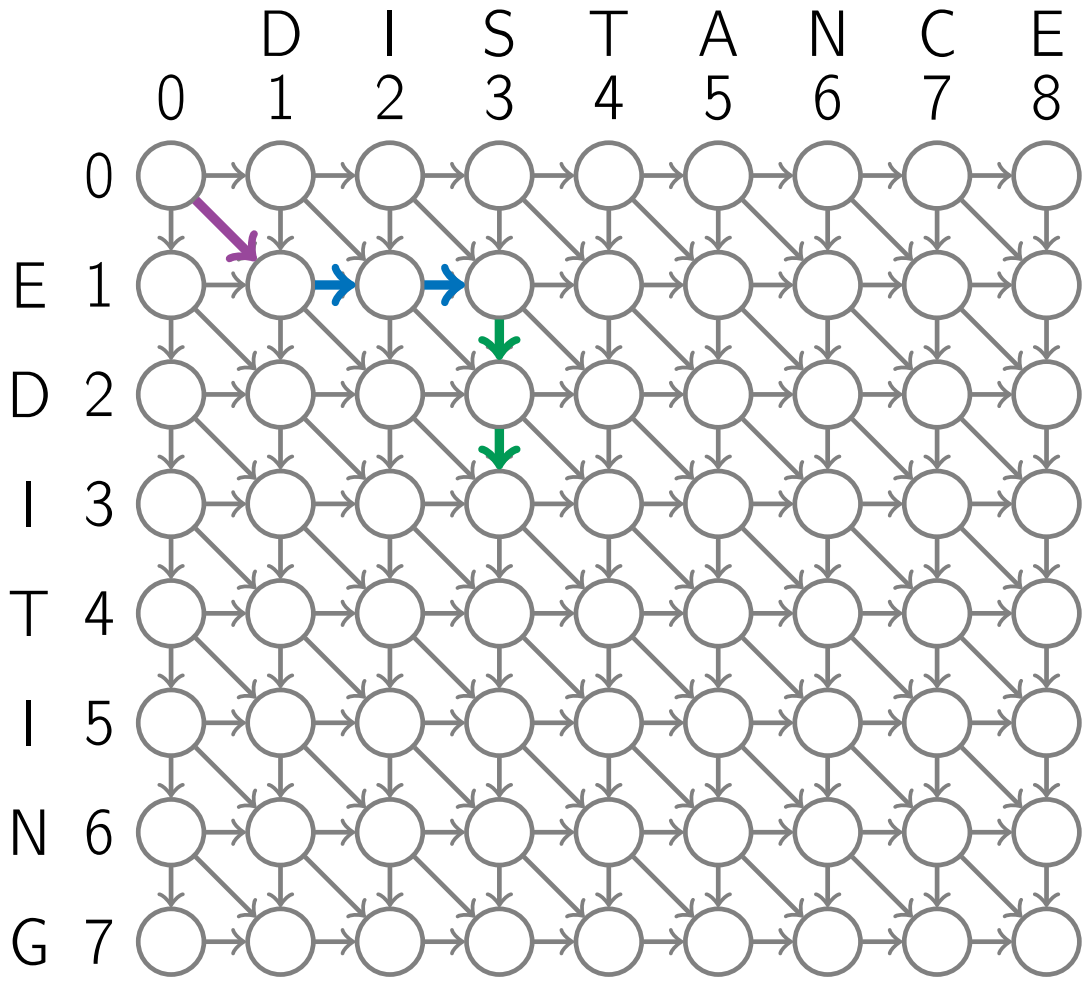
E	—	—
D	I	S

any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$



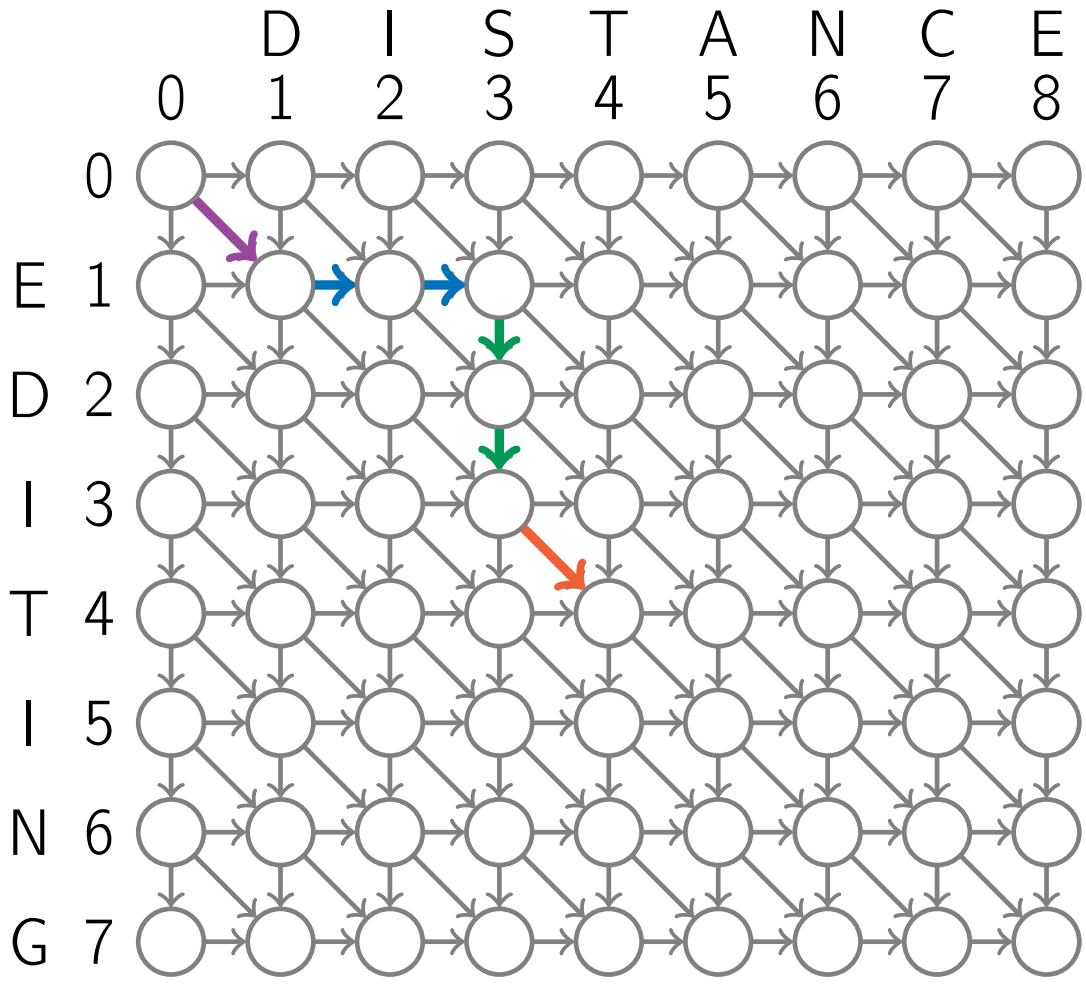
E	—	—	D
D	I	S	—

any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$



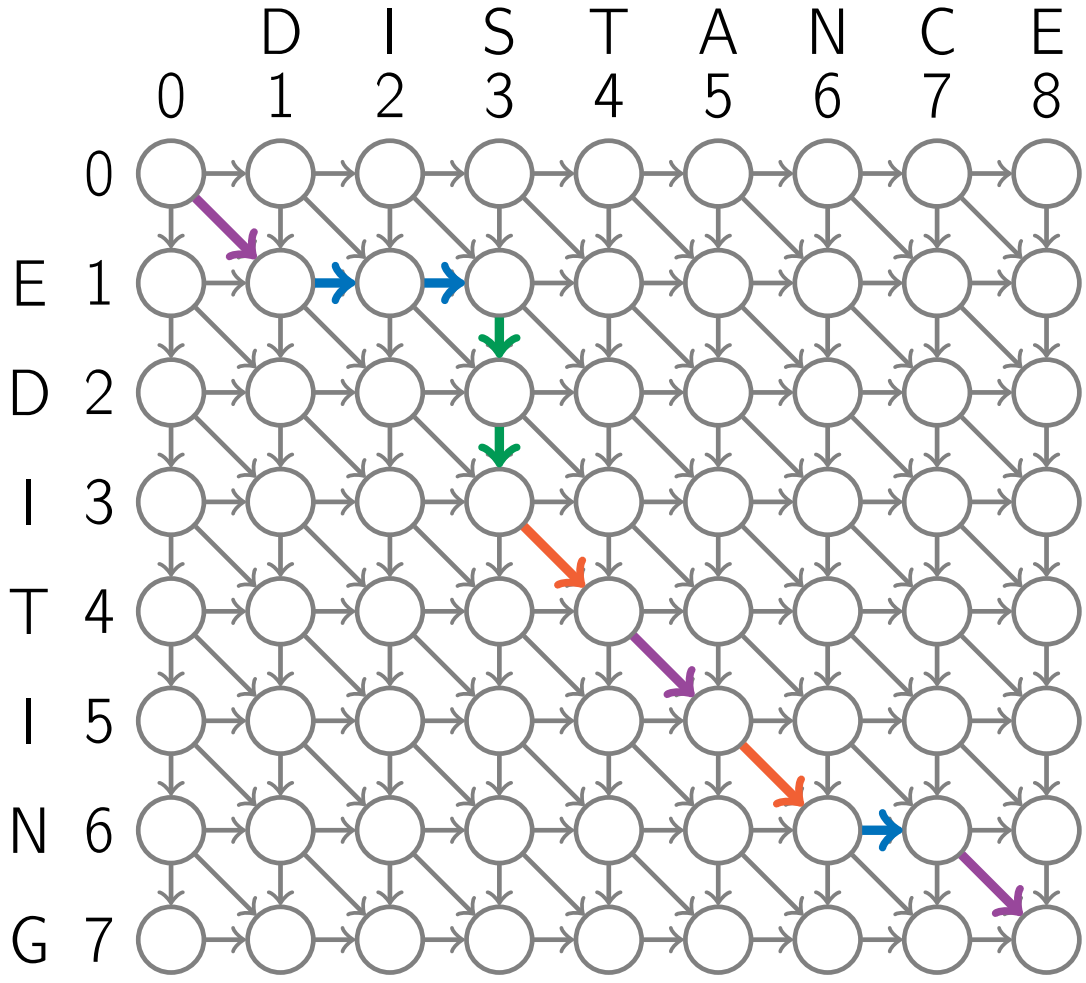
E	—	—	D	I
D	I	S	—	—

any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$



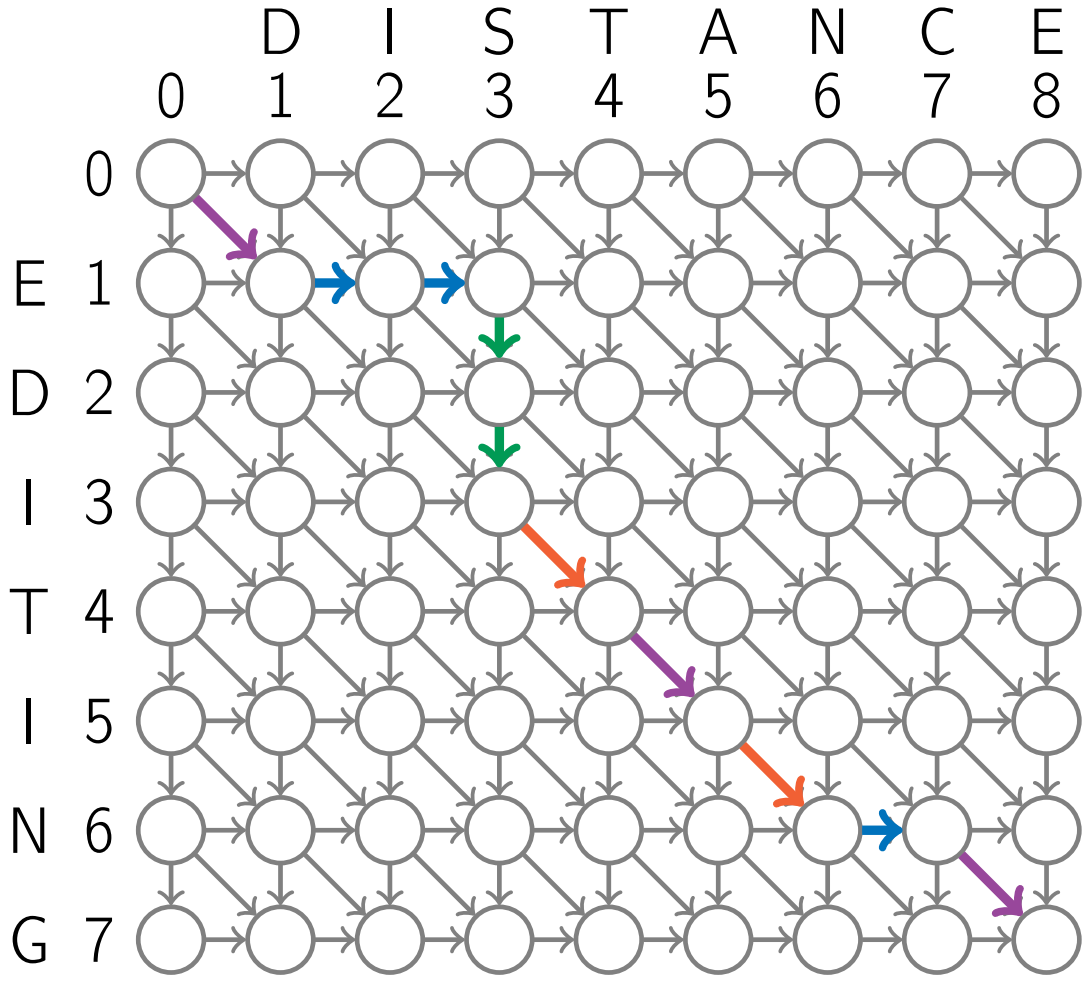
E	—	—	D	I	T
D	I	S	—	—	T

any path from $(0,0)$ to (i,j) spells an alignment of prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$



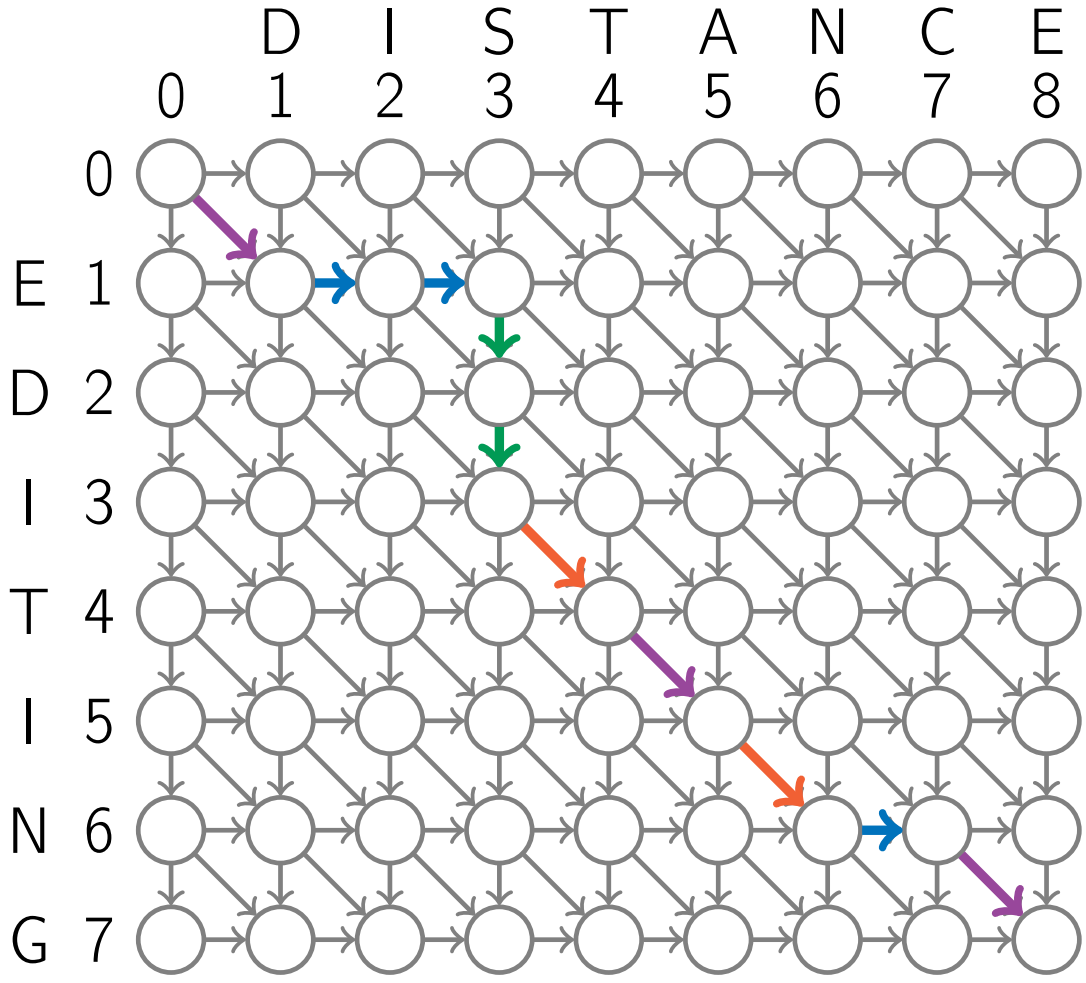
E	–	–	D	I	T	I	N	–	G
D	I	S	–	–	T	A	N	C	E

the constructed path corresponds to distance 8 and is not optimal (edit distance is 5)

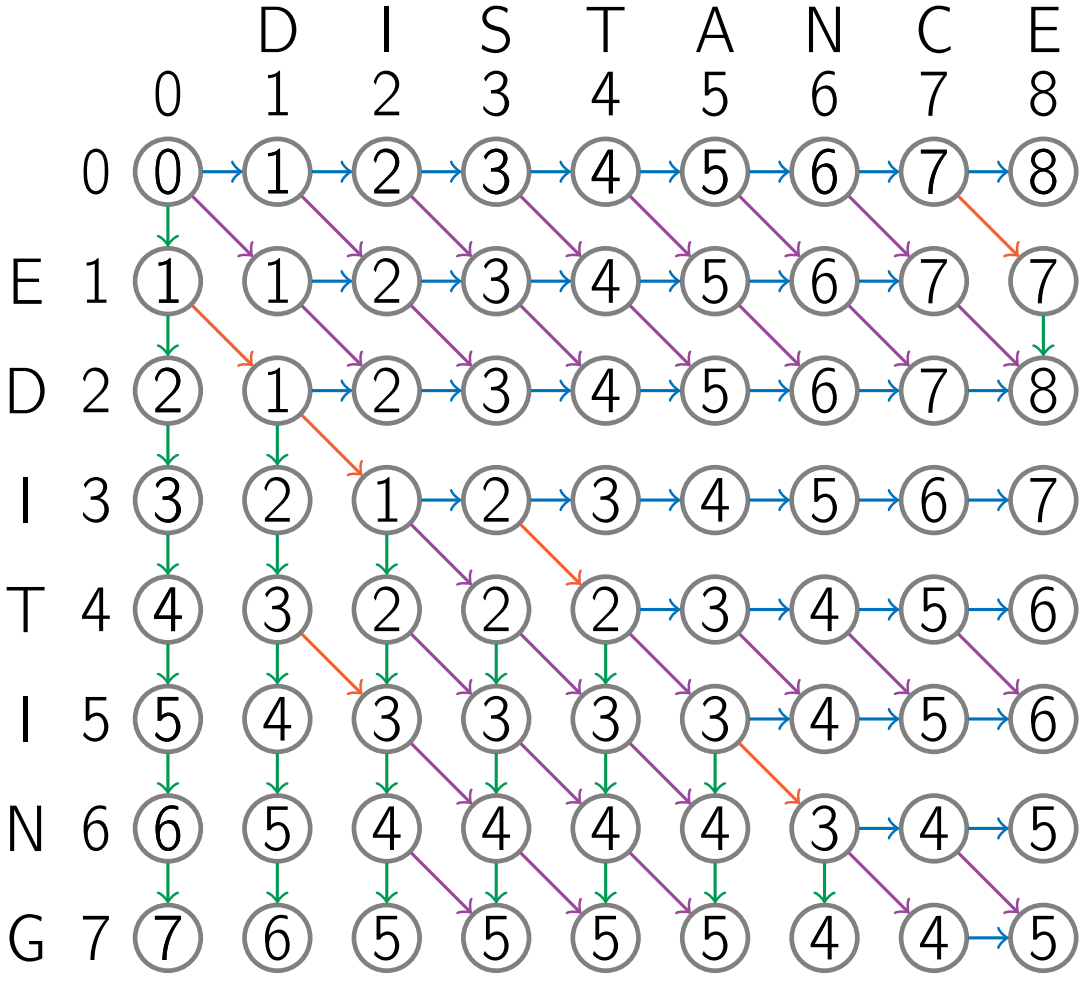


E	–	–	D	I	T	I	N	–	G
D	I	S	–	–	T	A	N	C	E

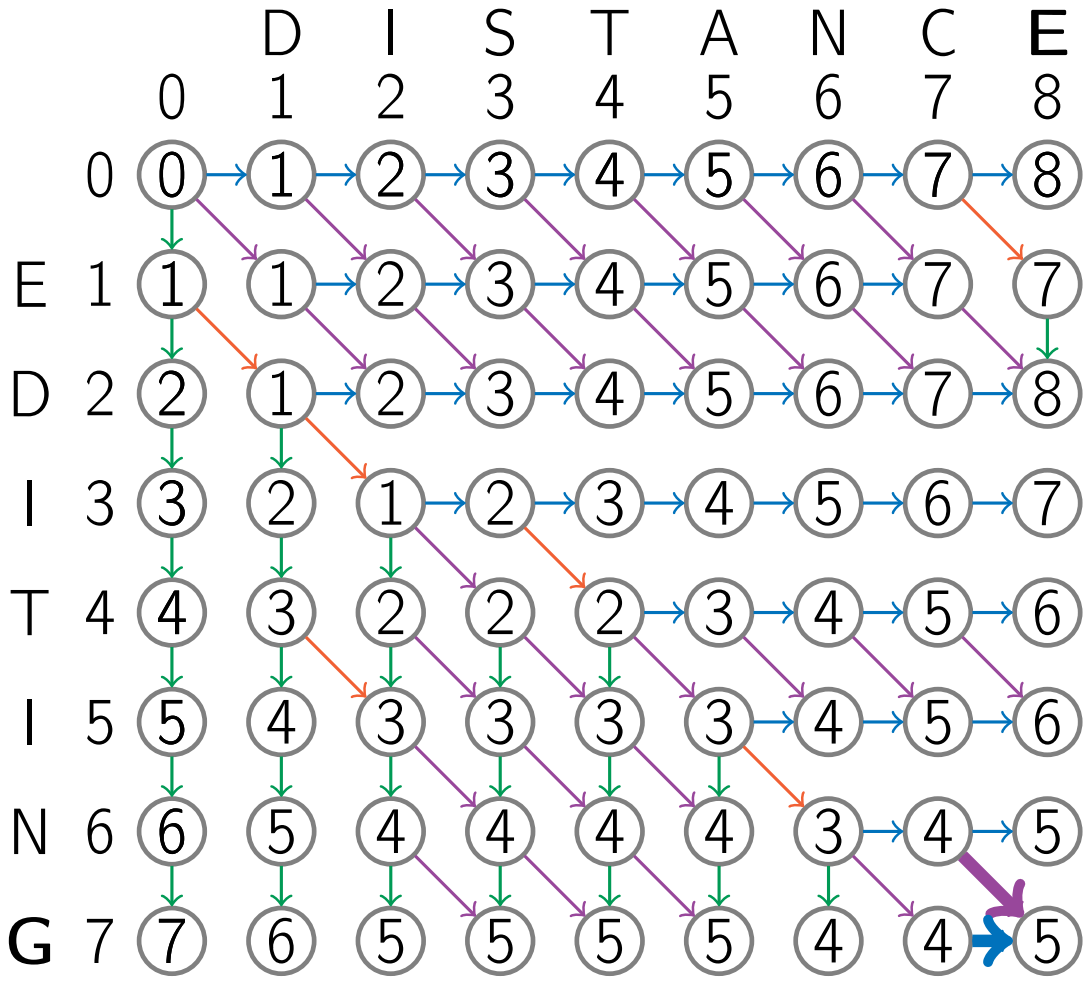
to construct an optimal alignment we will use the backtracking pointers



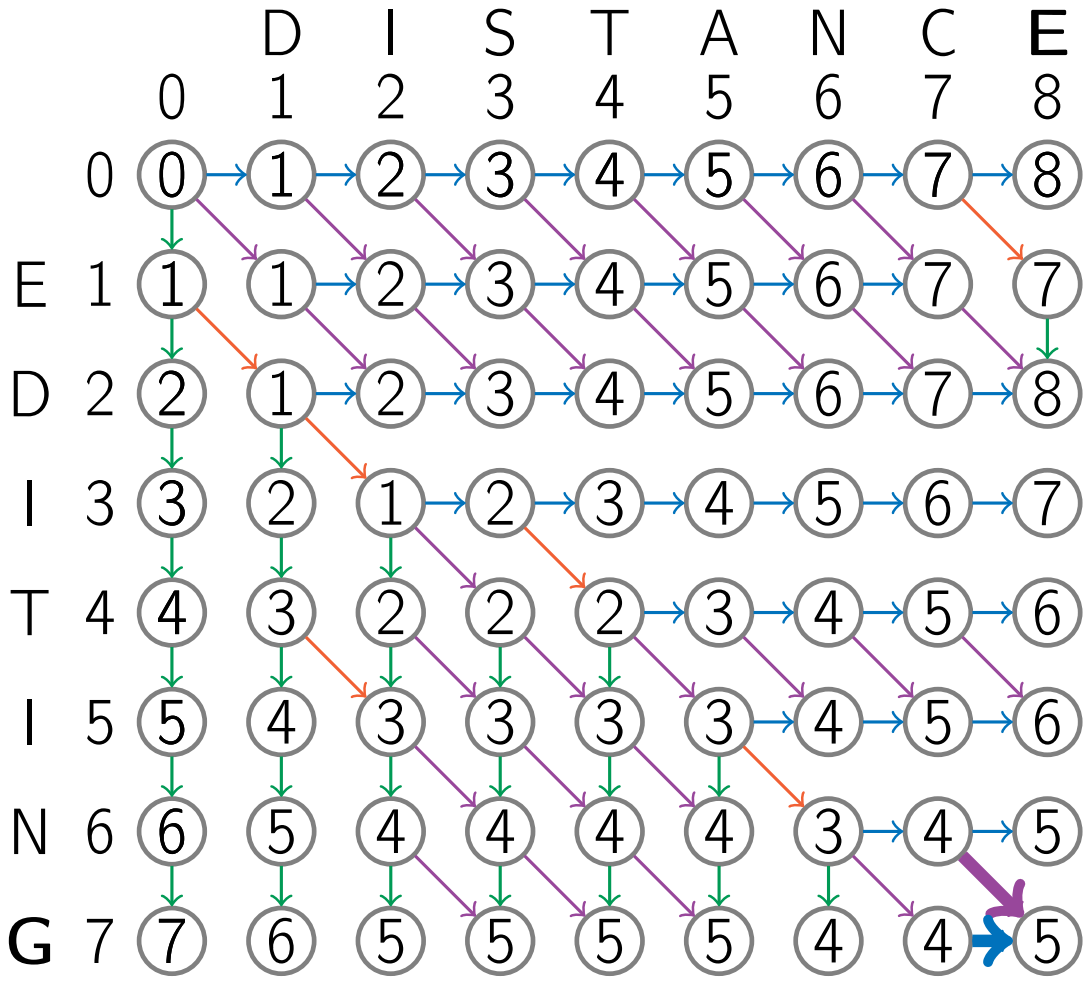
E	–	–	D	I	T	I	N	–	G
D	I	S	–	–	T	A	N	C	E



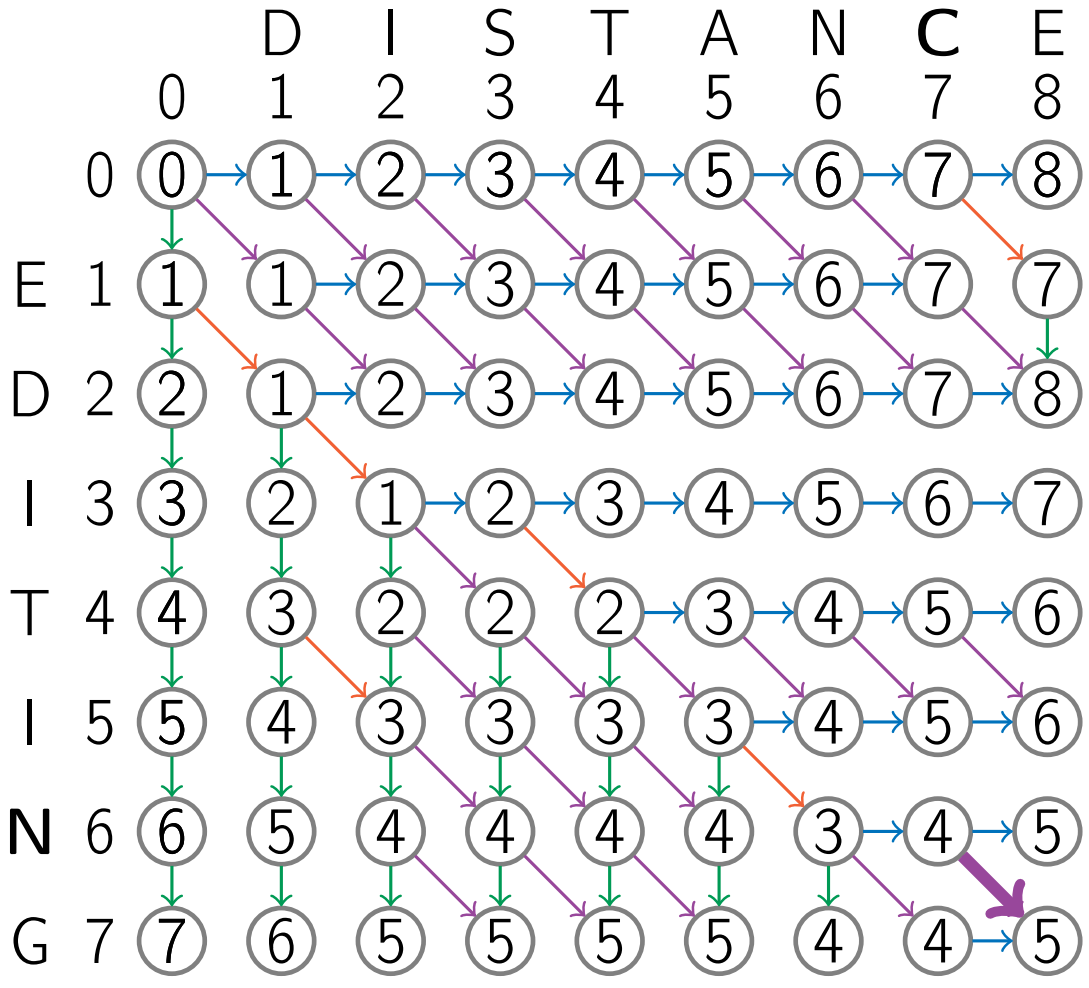
the edit distance is 5



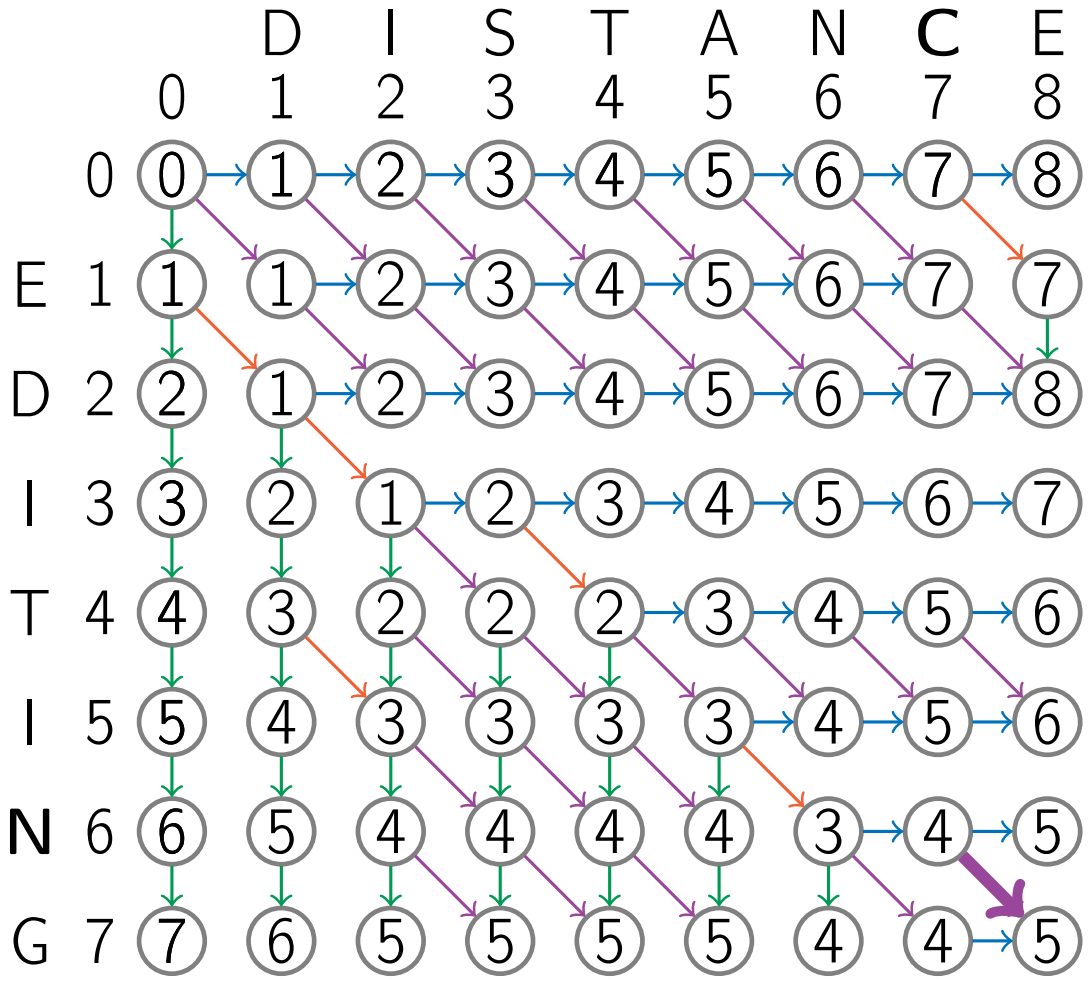
we arrived to the bottom right cell by moving along the backtracking pointers shown below



there exists an optimal alignment whose last column is a mismatch and an optimal alignment whose last column is an insertion

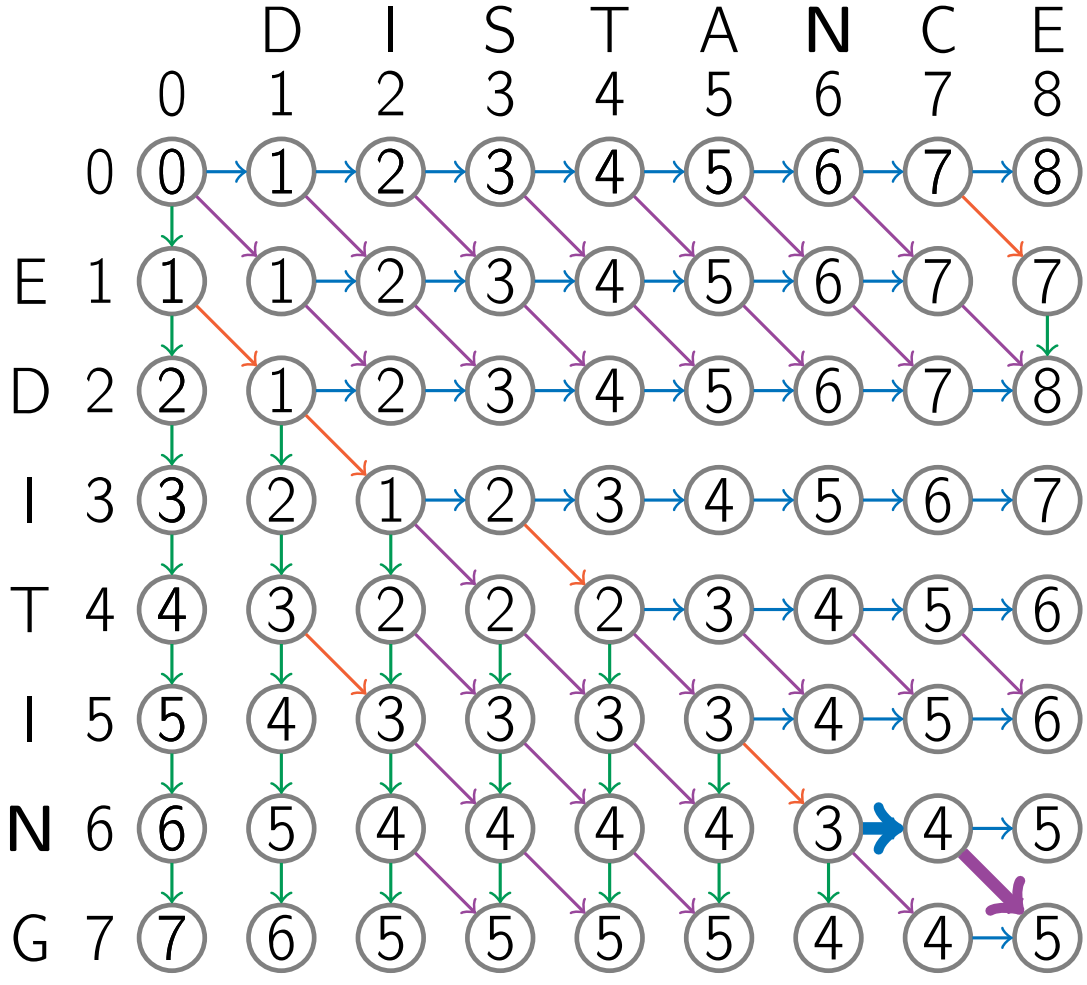


let's consider a mismatch

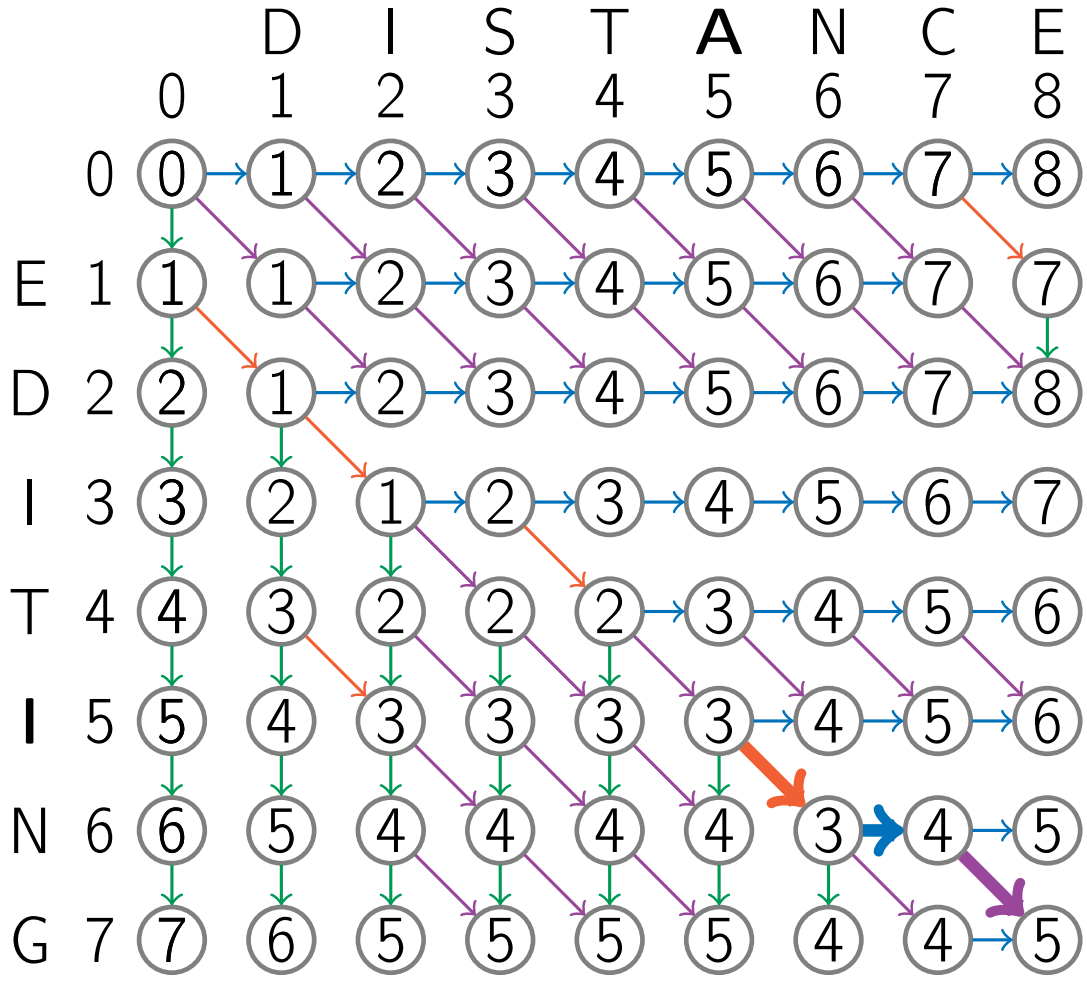


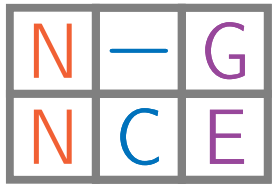
we continue in a similar fashion

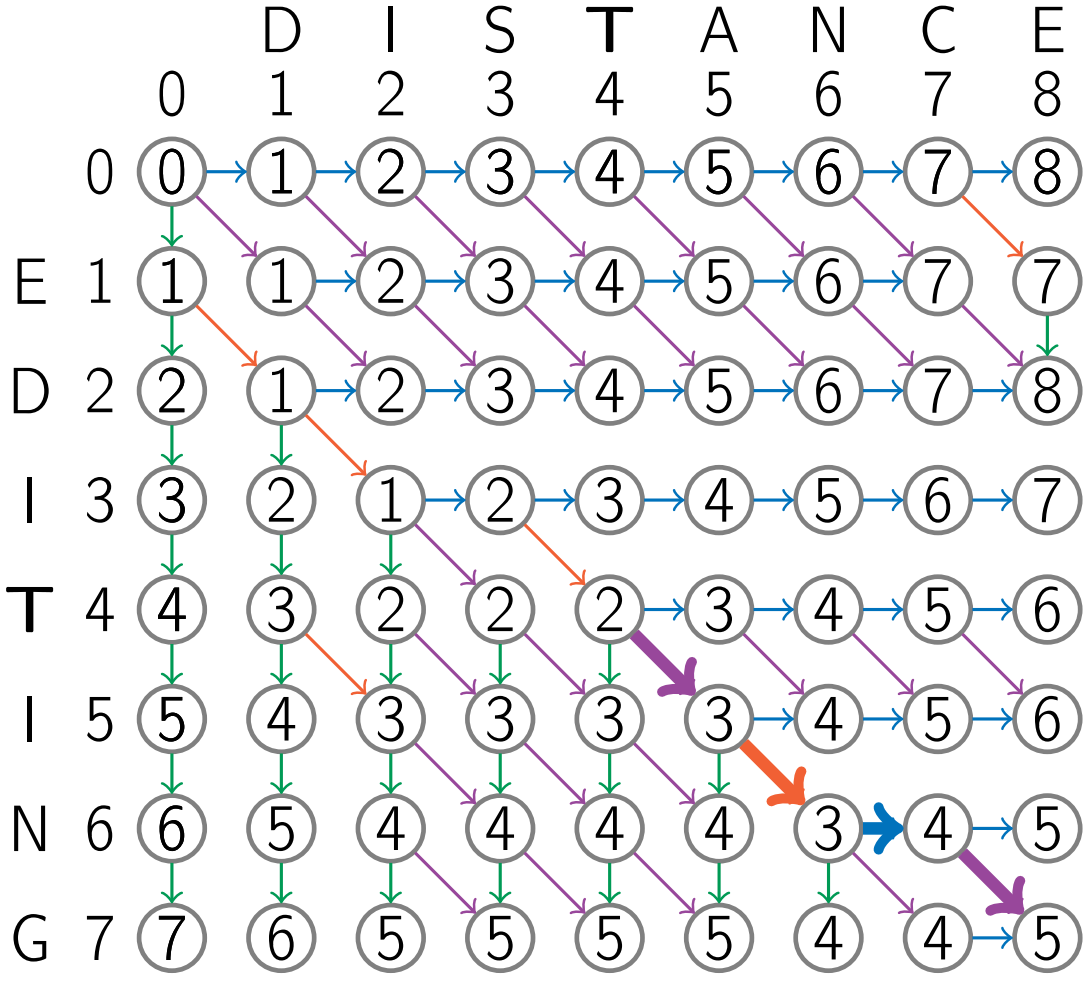
G
E


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1 2 3 4 5 6 7 7
D 2 2 1 2 3 4 5 6 7 8
I 3 3 2 1 2 3 4 5 6 7
T 4 4 3 2 2 2 3 4 5 6
I 5 5 4 3 3 3 3 4 5 6
N 6 6 5 4 4 4 4 3 4 5
G 7 7 6 5 5 5 5 4 4 5

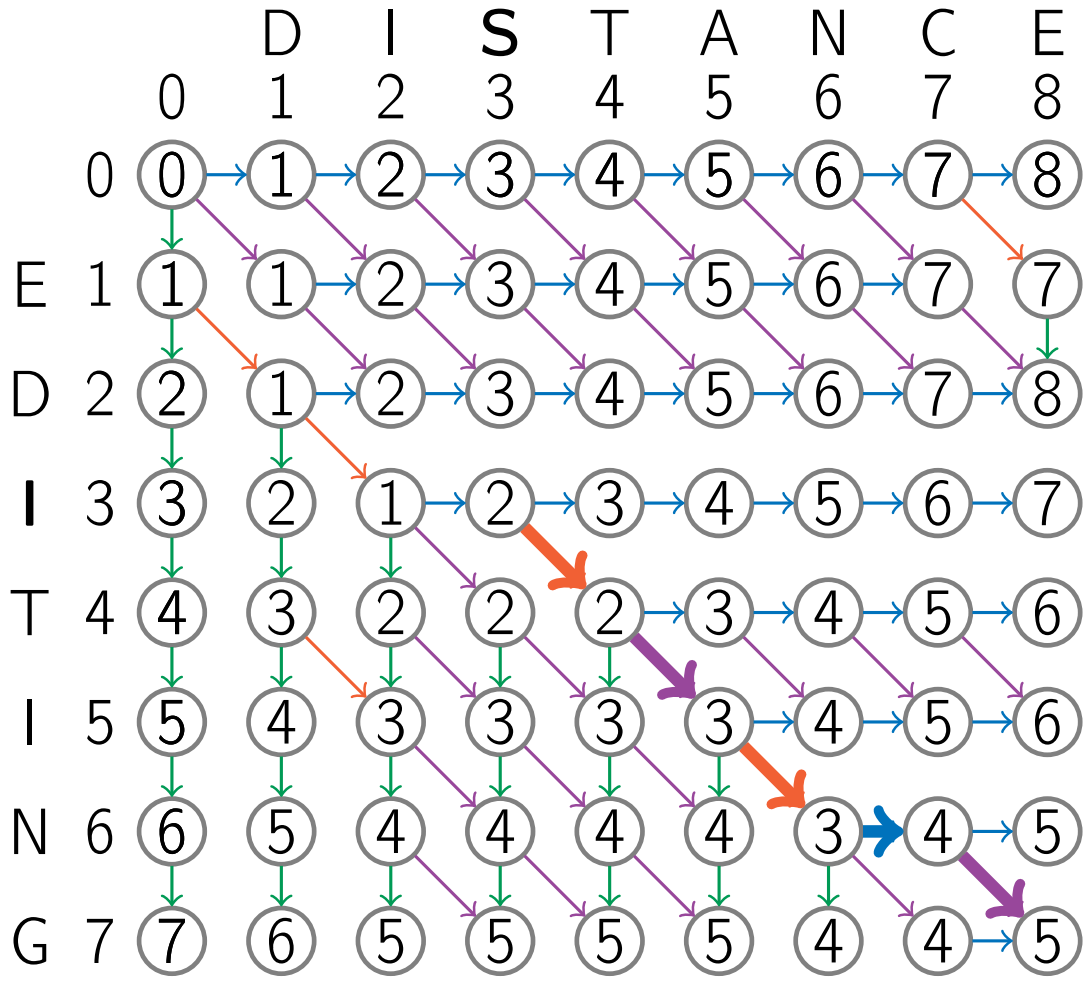

—
G
C
E


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1 2 3 4 5 6 7 7
D 2 2 1 2 3 4 5 6 7 8
I 3 3 2 1 2 3 4 5 6 7
T 4 4 3 2 2 2 3 4 5 6
I 5 5 4 3 3 3 3 4 5 6
N 6 6 5 4 4 4 4 3 4 5
G 7 7 6 5 5 5 5 4 4 5

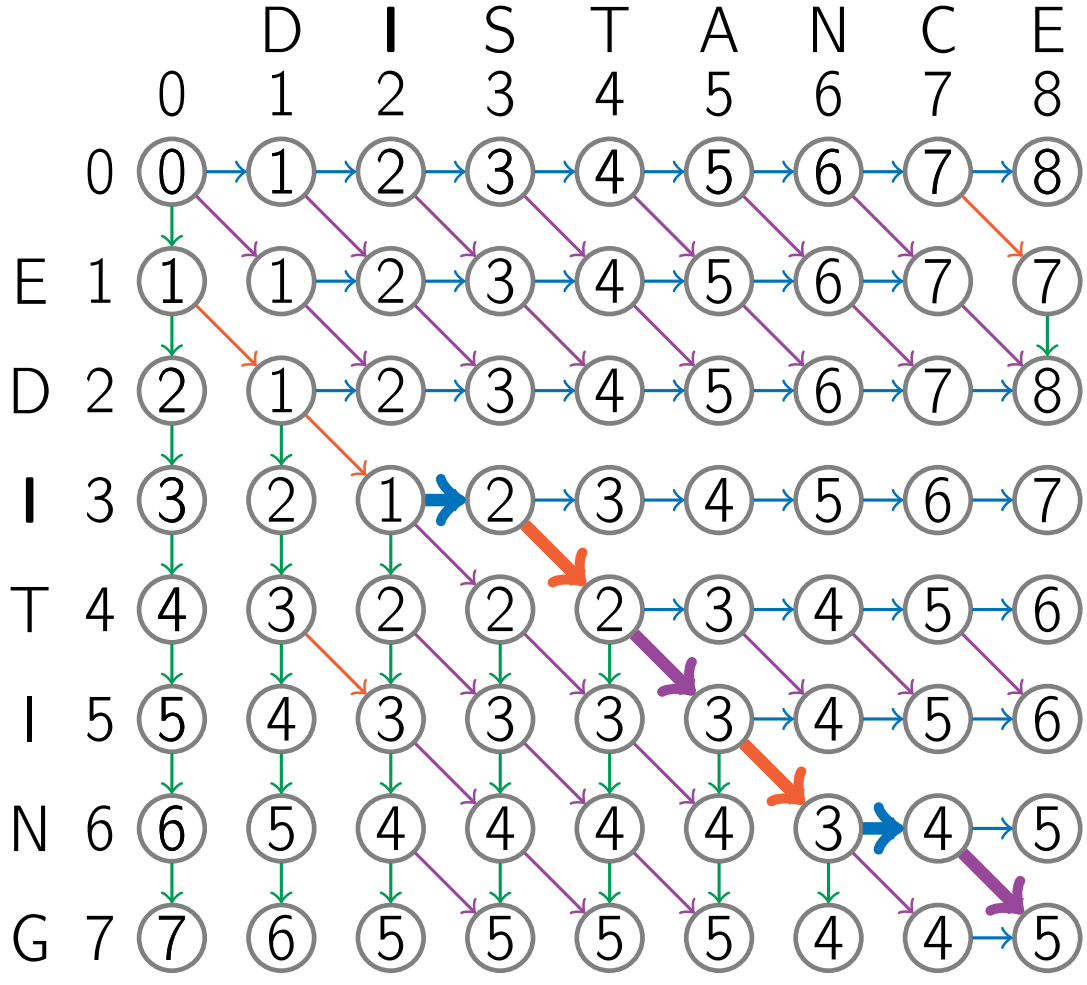

N — G
N C E


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1 2 3 4 5 6 7 7
D 2 2 1 2 3 4 5 6 7 8
I 3 3 2 1 2 3 4 5 6 7
T 4 4 3 2 2 2 3 4 5 6
I 5 5 4 3 3 3 3 4 5 6
N 6 6 5 4 4 4 4 3 4 5
G 7 7 6 5 5 5 5 4 4 5

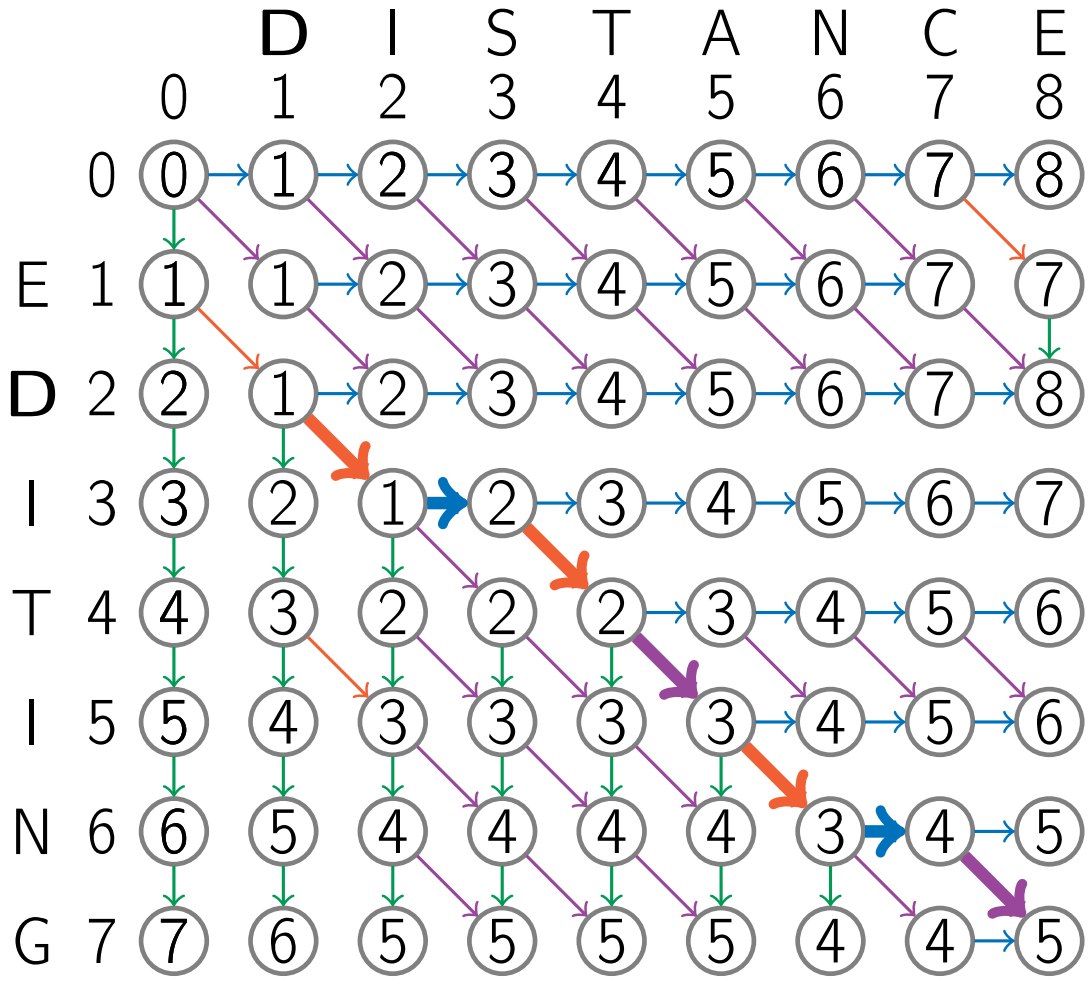
I	N	—	G
A	N	C	E


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 0 1 2 3 4 5 6 7 8
E 1 1 1 2 3 4 5 6 7 7
D 2 2 1 2 3 4 5 6 7 8
I 3 3 2 1 2 3 4 5 6 7
T 4 4 3 2 2 2 3 4 5 6
I 5 5 4 3 3 3 3 4 5 6
N 6 6 5 4 4 4 4 3 4 5
G 7 7 6 5 5 5 5 4 4 5

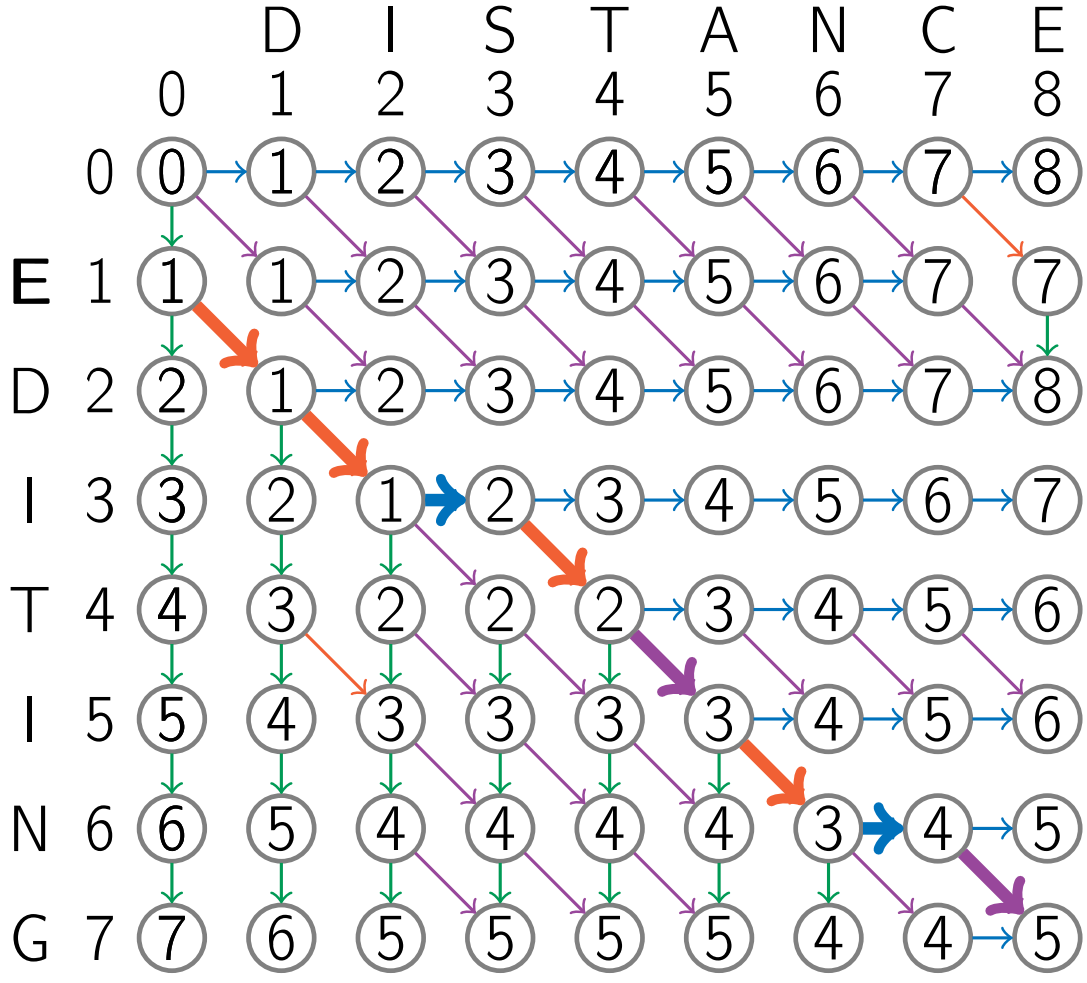
T	I	N	—	G
T	A	N	C	E


D I S T A N C E
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
E 1 1 2 3 4 5 6 7 7
D 2 2 1 2 3 4 5 6 7 8
I 3 3 2 1 2 3 4 5 6 7
T 4 4 3 2 2 2 3 4 5 6
I 5 5 4 3 3 3 3 4 5 6
N 6 6 5 4 4 4 4 3 4 5
G 7 7 6 5 5 5 5 4 4 5

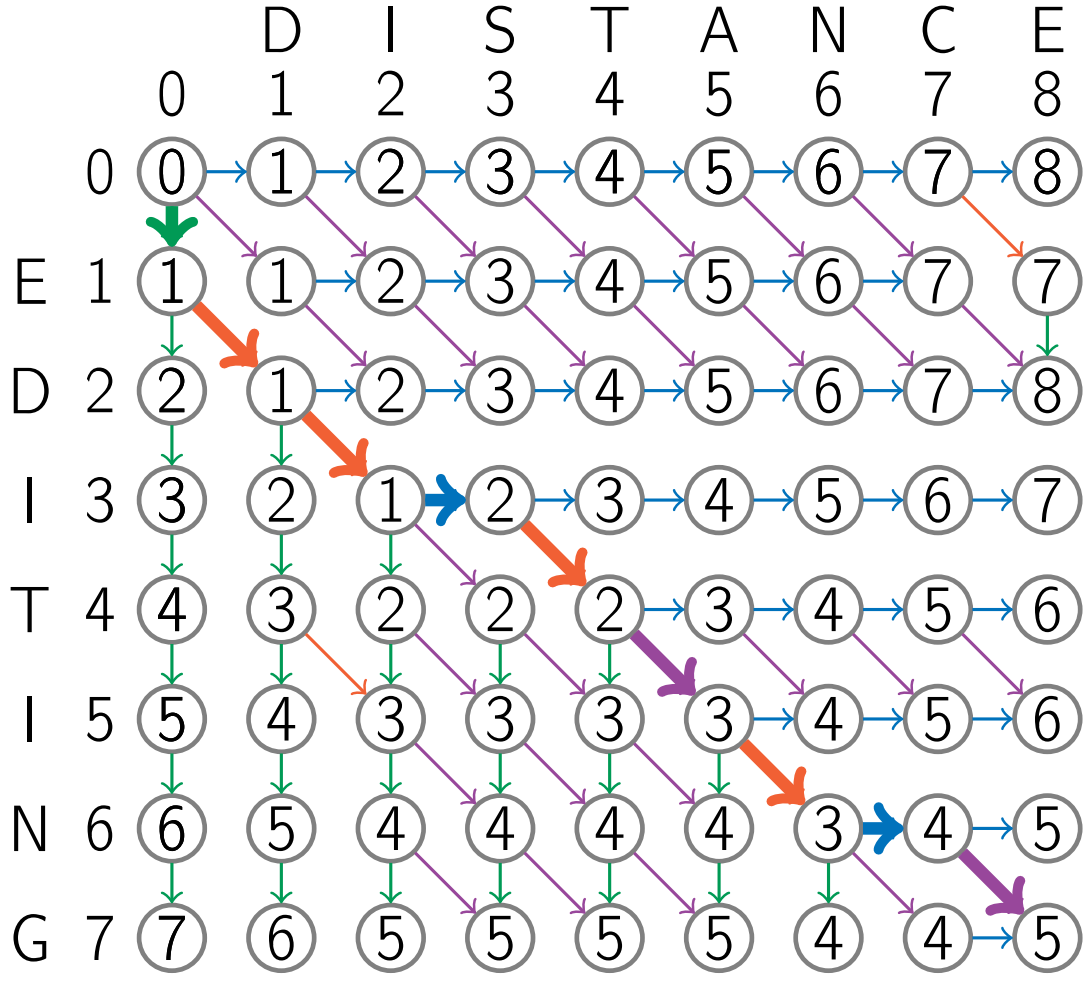
—	T	I	N	—	G
S	T	A	N	C	E



I	—	T	I	N	—	G
I	S	T	A	N	C	E



D	I	—	T	I	N	—	G
D	I	S	T	A	N	C	E



E	D	I	—	T	I	N	—	G
—	D	I	S	T	A	N	C	E

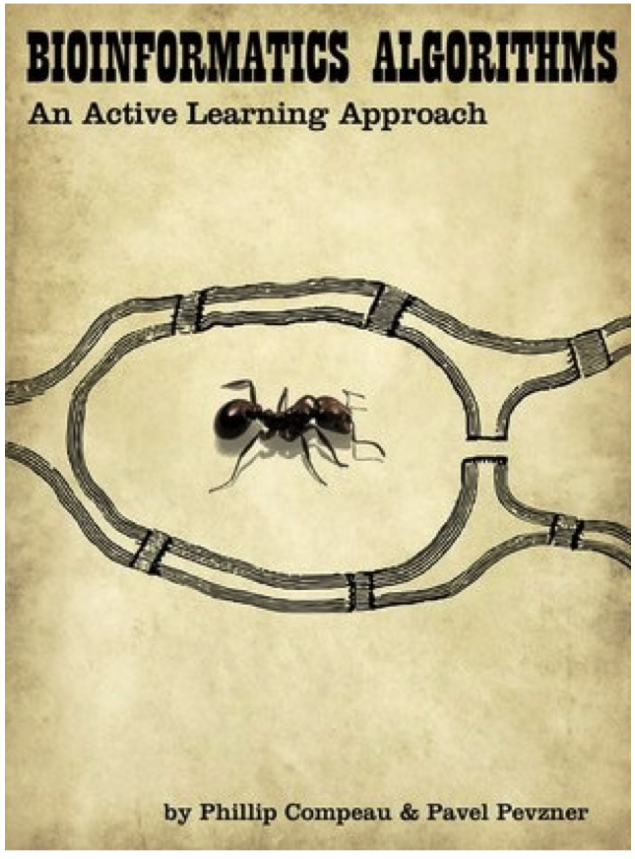
OutputAlignment(i, j)

```
if i = 0 and j = 0:
    return
if backtrack(i, j) = ↓:
    OutputAlignment(i − 1, j)
    print A[i] / −
else if backtrack(i, j) = →:
    OutputAlignment(i, j − 1)
    print − / B[j]
else:
    OutputAlignment(i − 1, j − 1)
    print A[i] / B[j]
```

OutputAlignment(i, j)

```
if i = 0 and j = 0:
    return
if i > 0 and D(i, j) = D(i − 1, j) + 1:
    OutputAlignment(i − 1, j)
    print A[i] / −
else if j > 0 and D(i, j) = D(i, j − 1) + 1:
    OutputAlignment(i, j − 1)
    print − / B[j]
else:
    OutputAlignment(i − 1, j − 1)
    print A[i] / B[j]
```

Comparing Genes, Proteins, and Genomes MOOC (a part of Bioinformatics Specialization on Coursera)



Bioinformatics Algorithms textbook at bioinformaticsalgorithms.org (2nd two-volume edition was published in 2015)

Dynamic Programming: Knapsack

Alexander S. Kulikov

St. Petersburg Department of Steklov Institute of Mathematics
Russian Academy of Sciences

Data Structures and Algorithms
Algorithmic Toolbox

Outline

TV commercial placement

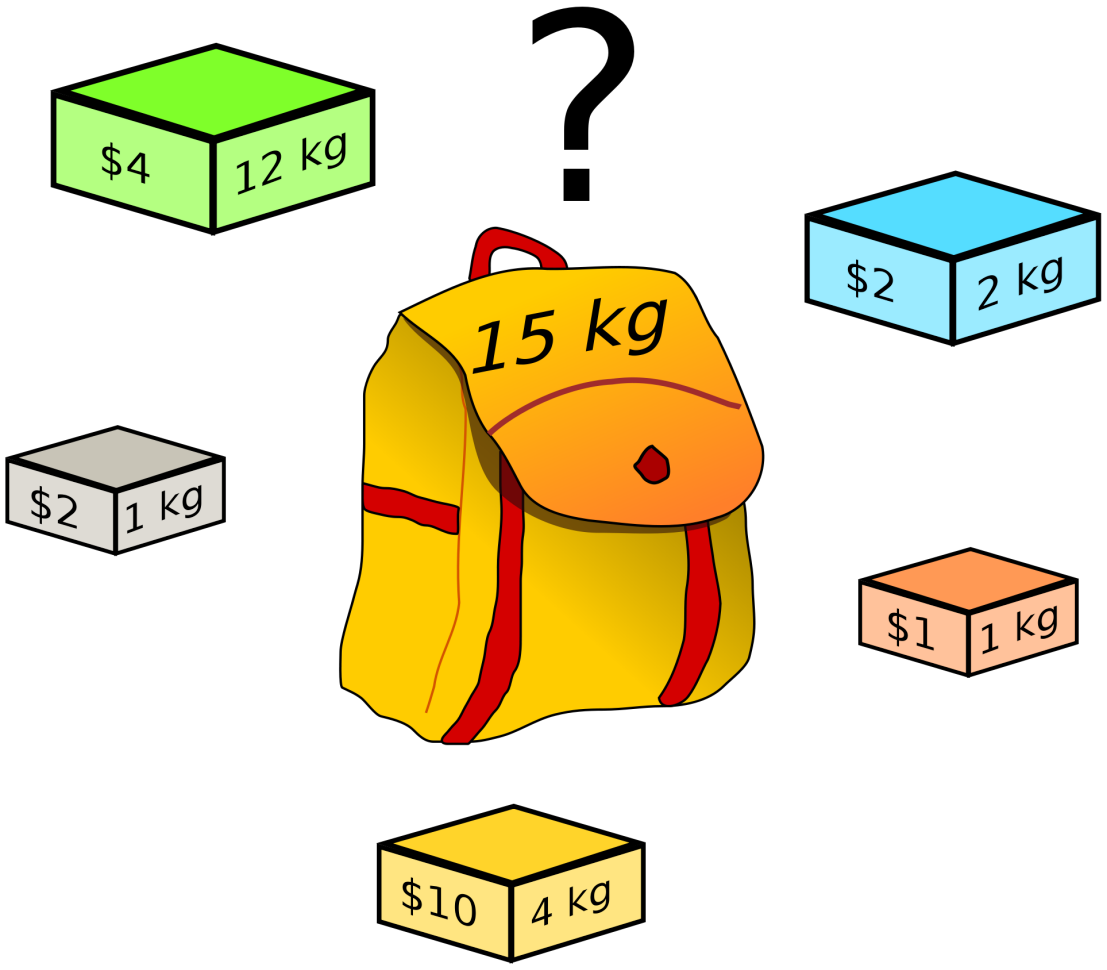
Select a set of TV commercials (each commercial has duration and cost) so that the total revenue is maximal while the total length does not exceed the length of the available time slot.

Optimizing data center performance

Purchase computers for a data center to achieve the maximal performance under limited budget.

Knapsack Problem

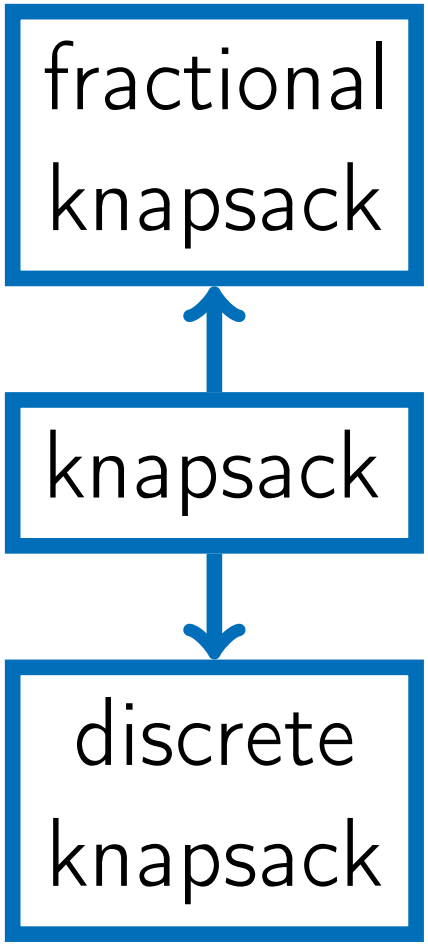
(knapsack is another word for backpack)



Goal

Maximize value ($) while limiting total weight (kg)

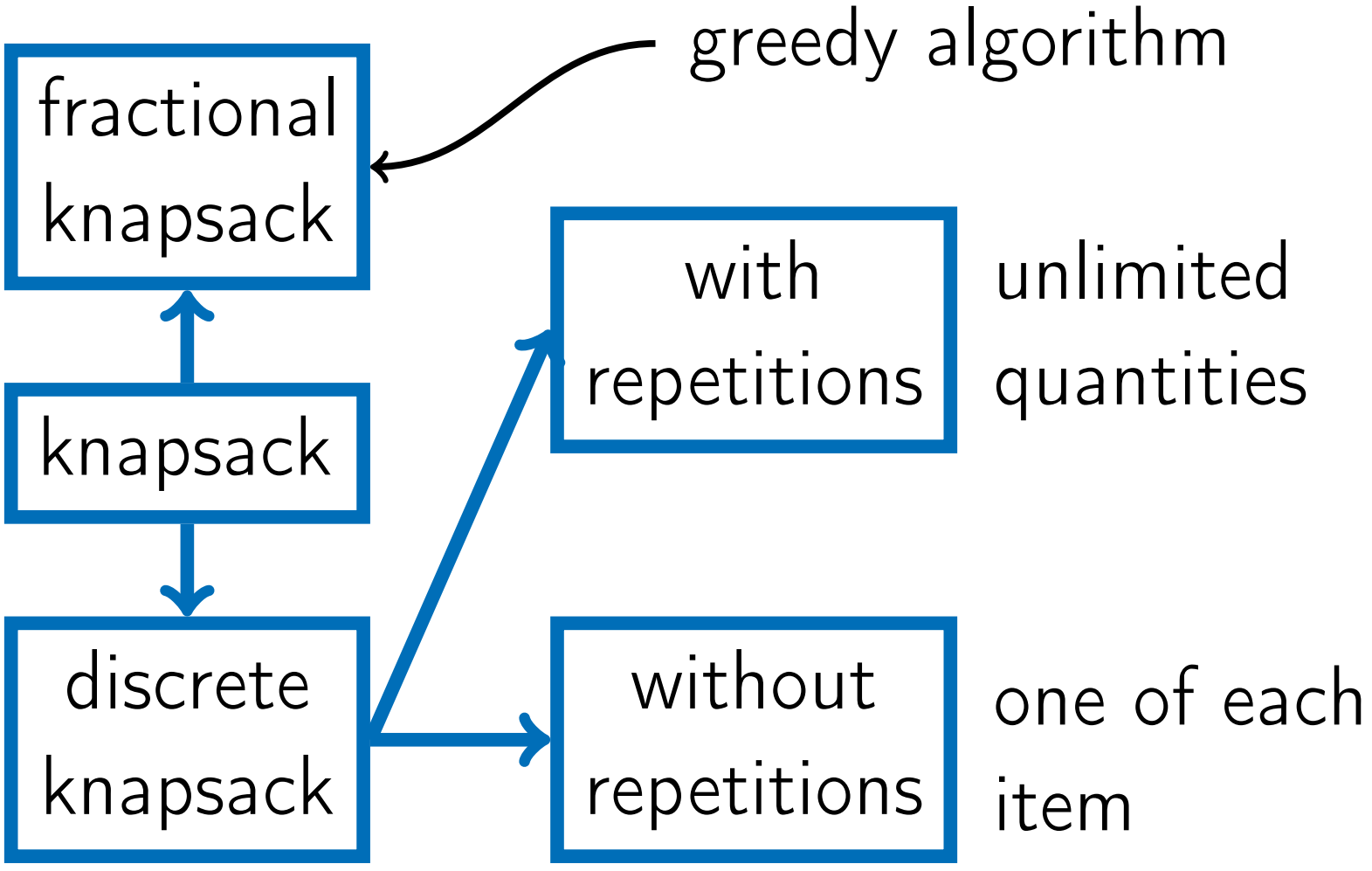
Problem Variations



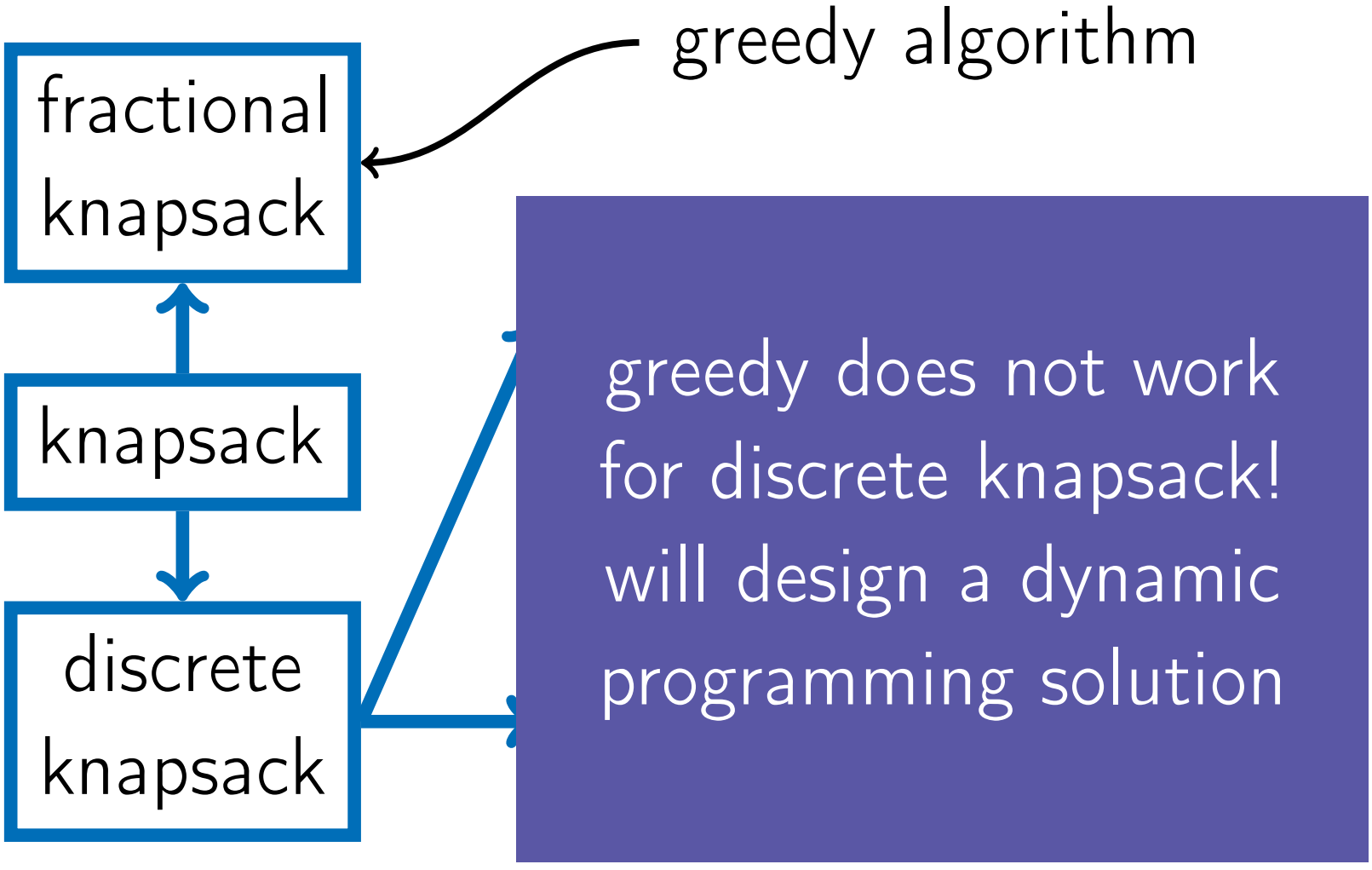
can take fractions of items

each item is either taken or not

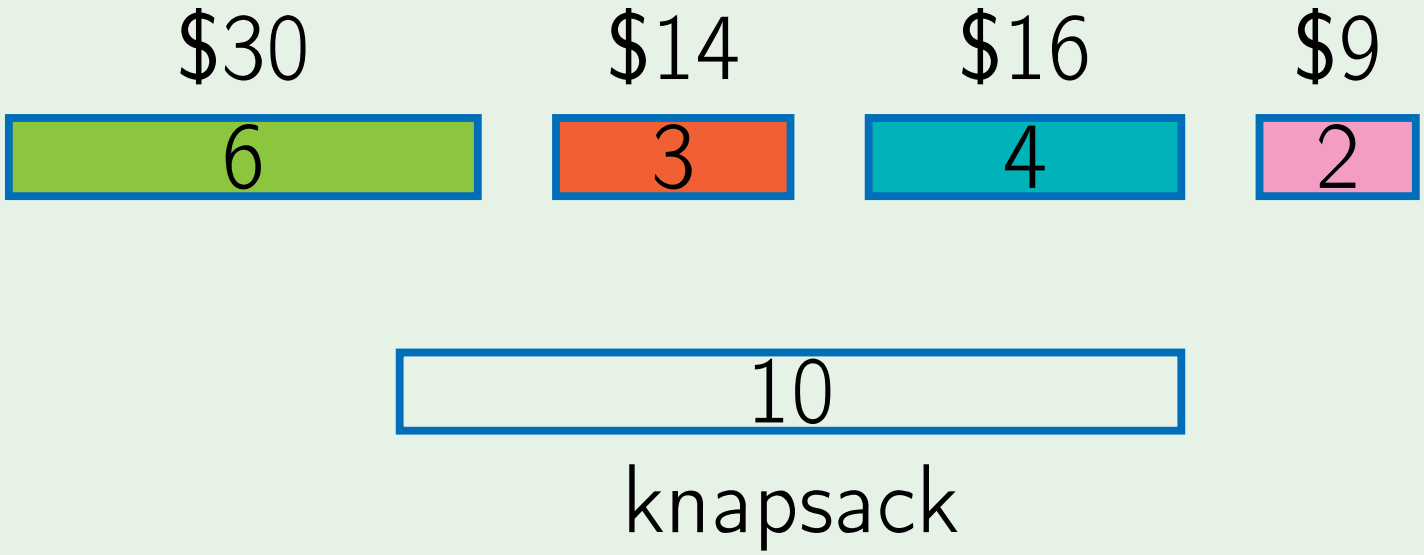
Problem Variations



Problem Variations

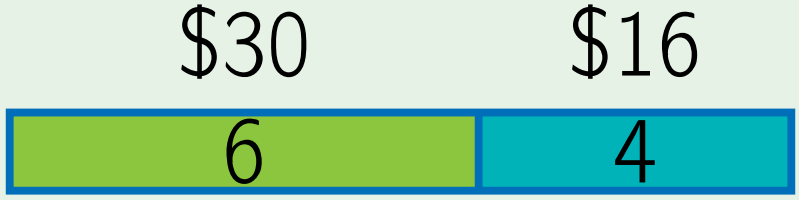


Example



Example

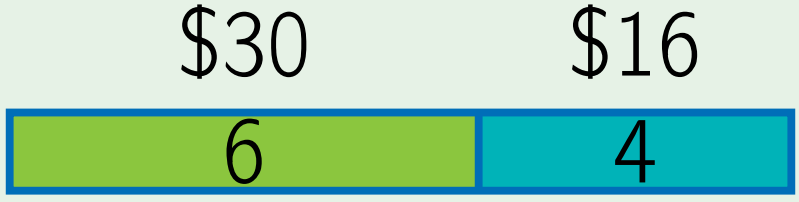
w/o repeats



total: $46

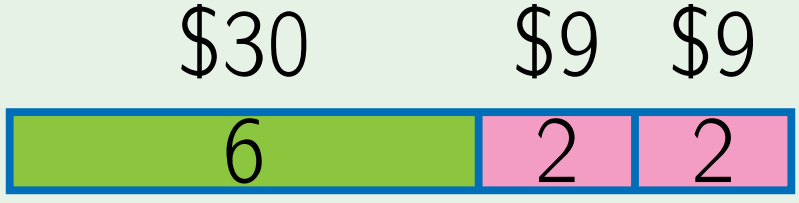
Example

w/o repeats



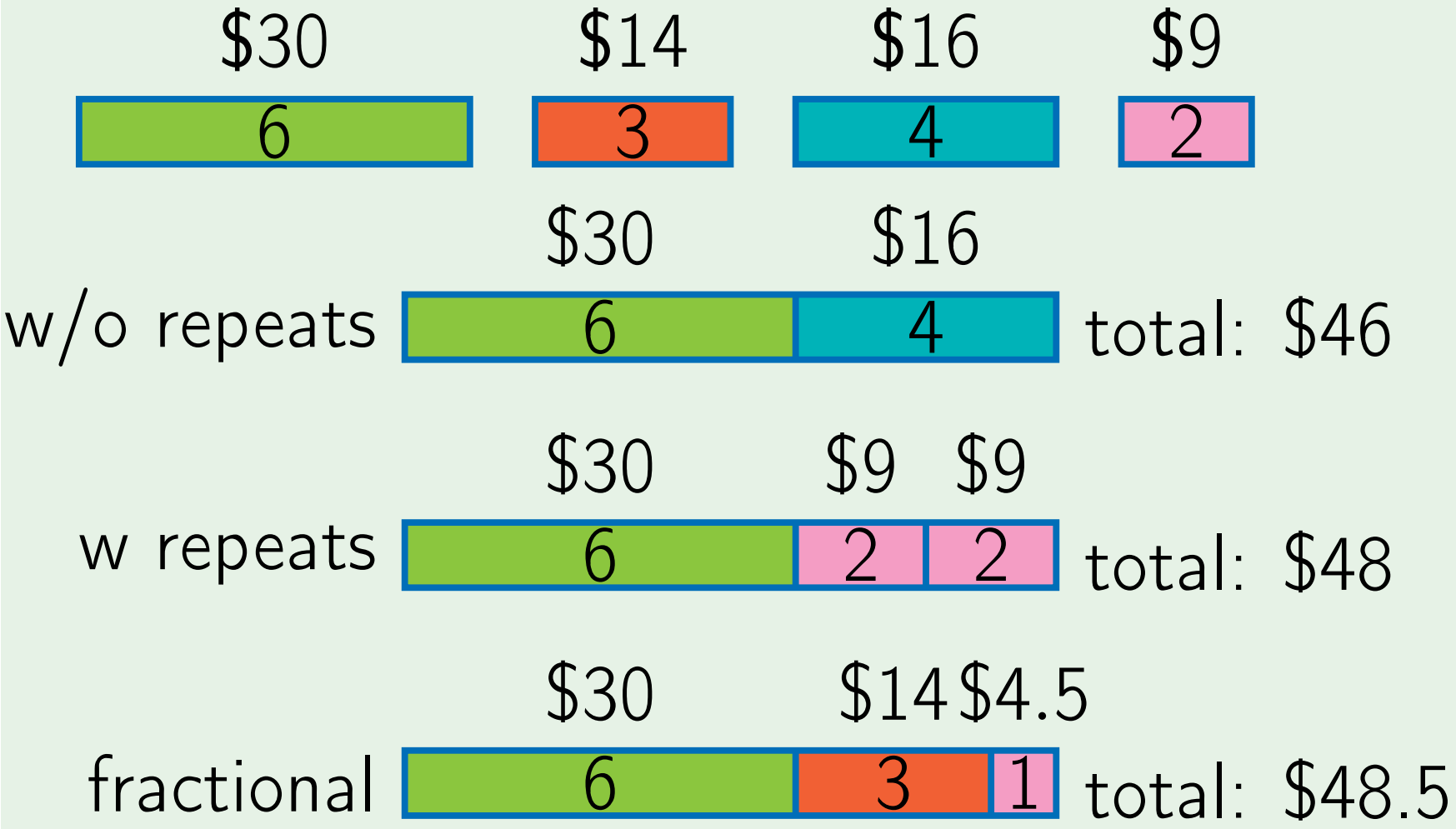
total: $46

w repeats



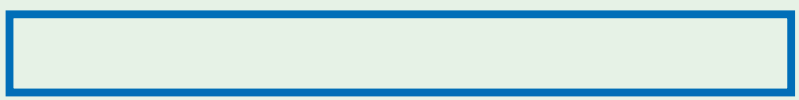
total: $48

Example


$30
$14
$16
$9
6
3
4
2
$30
$16
w/o repeats
6
4
total: $46
$30
$9
$9
w repeats
6
2
2
total: $48
$30
$14
$4.5
fractional
6
3
1
total: $48.5

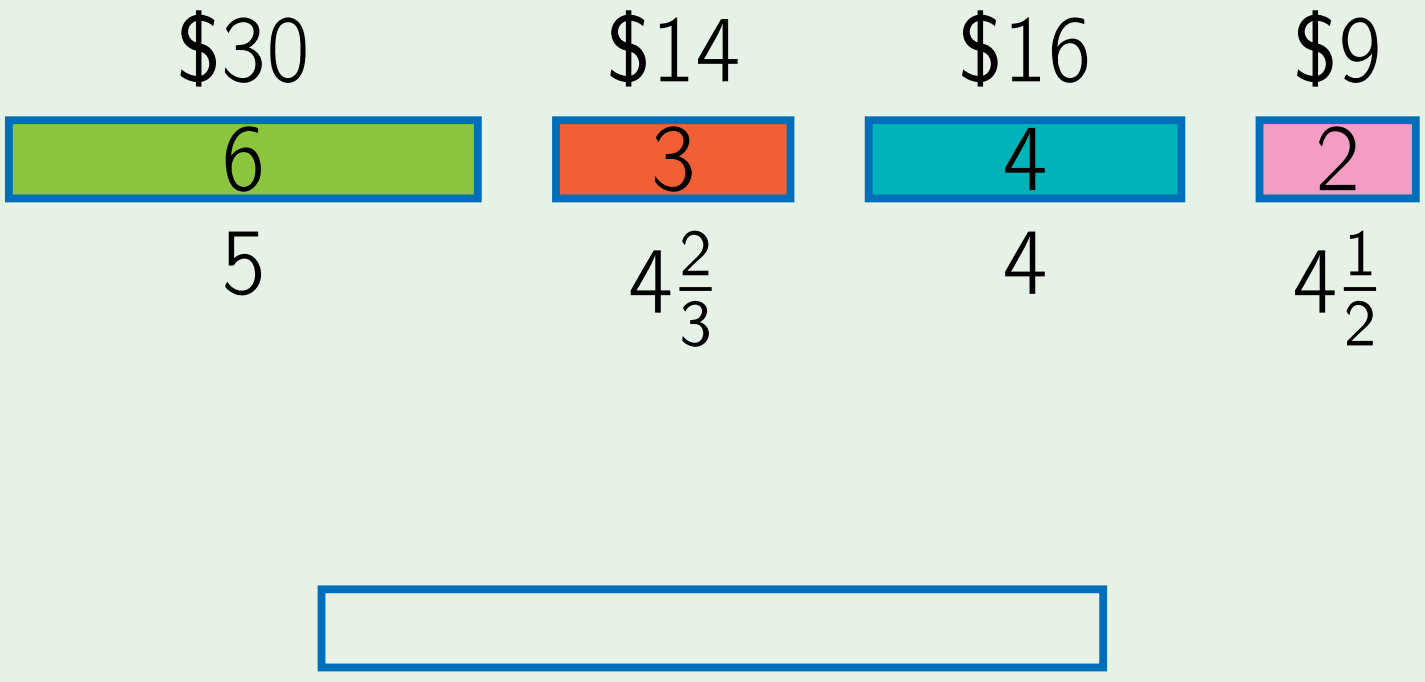
Why does greedy fail for the discrete knapsack?

Example



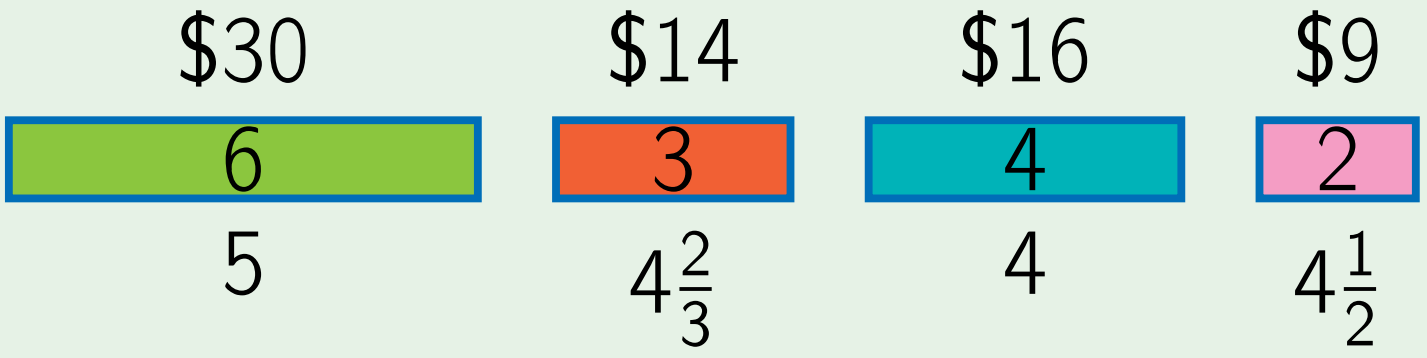
Why does greedy fail for the discrete knapsack?

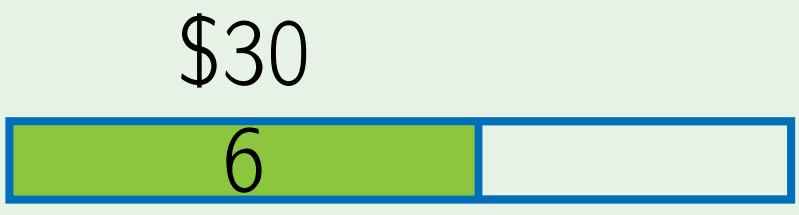
Example



Why does greedy fail for the discrete knapsack?

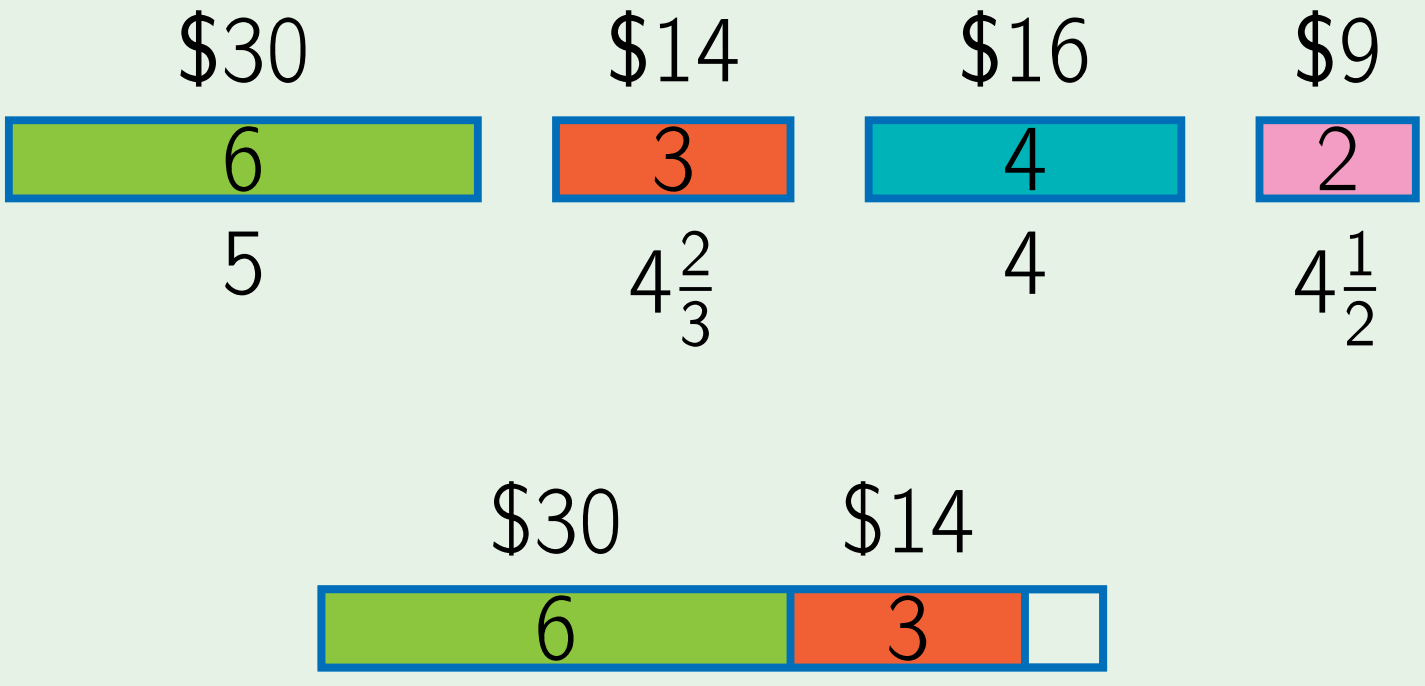
Example





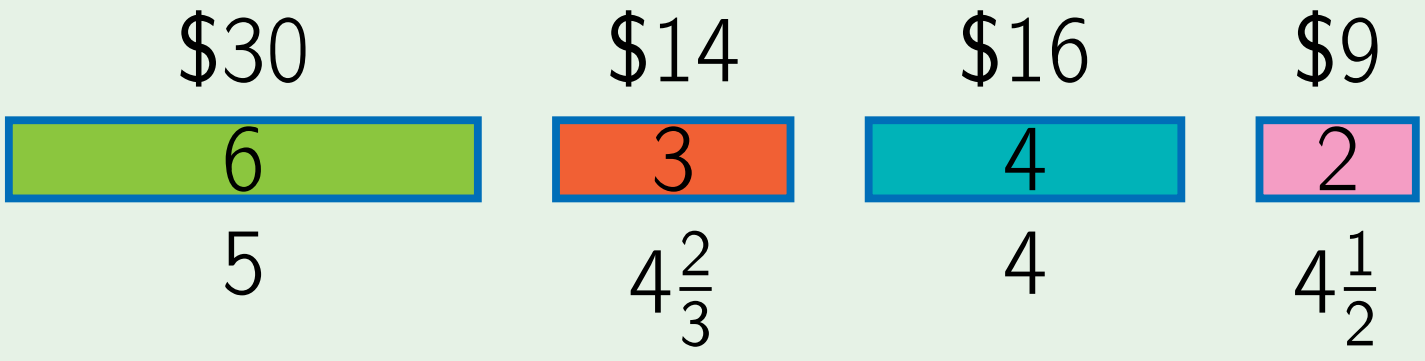
Why does greedy fail for the discrete knapsack?

Example



Why does greedy fail for the discrete knapsack?

Example

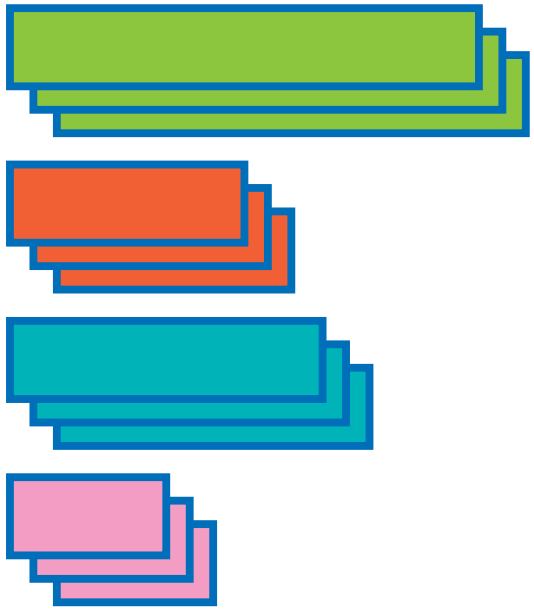


taking an element of maximum value per unit of weight is not safe!

Outline

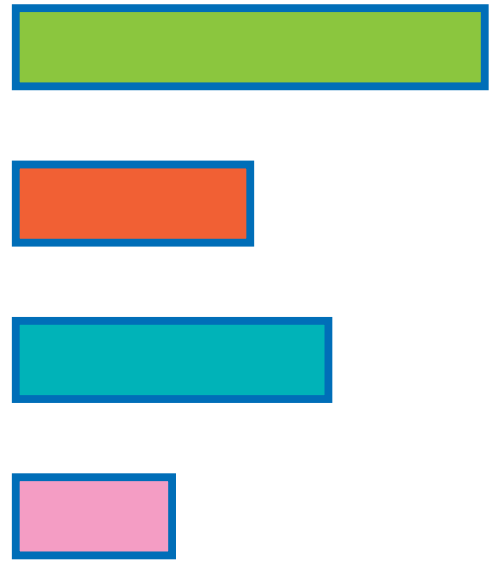
With repetitions:

unlimited quantities



Without repetitions:

one of each item



Knapsack with repetitions problem

Input: Weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ of n items; total weight W (v_i's, w_i's, and W are non-negative integers).

Output: The maximum value of items whose weight does not exceed W. Each item can be used any number of times.

Subproblems

- Consider an optimal solution and an item in it:

- If we take this item out then we get an optimal solution for a knapsack of total weight $W - w_i$.

Subproblems

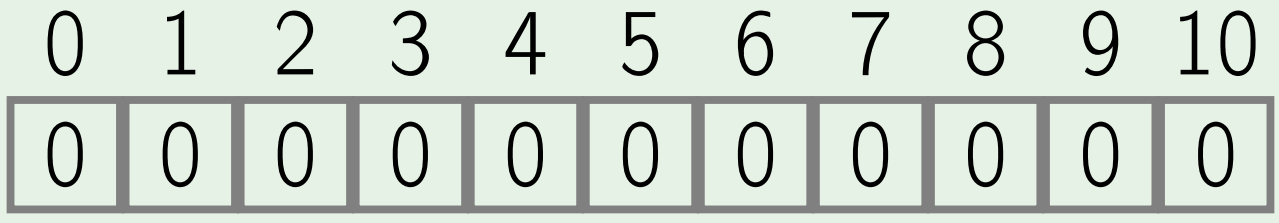
Let $value(w)$ be the maximum value of knapsack of weight w.

$$value(w) = \max_{i\,:\,w_i \le w} \{value(w - w_i) + v_i\}$$

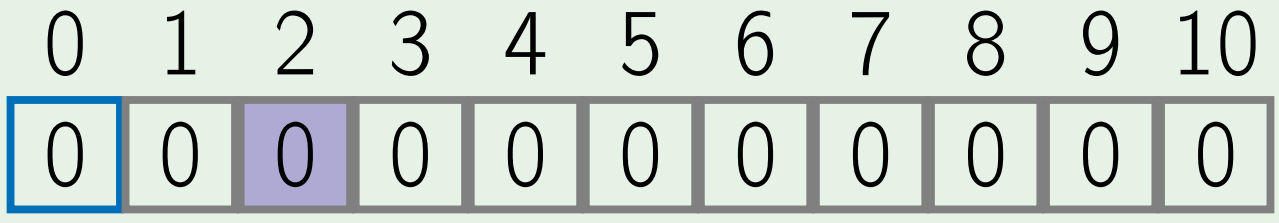
Knapsack(W)

```
value(0) ← 0
for w from 1 to W:
    value(w) ← 0
    for i from 1 to n:
        if w_i ≤ w:
            val ← value(w − w_i) + v_i
            if val > value(w):
                value(w) ← val
return value(W)
```

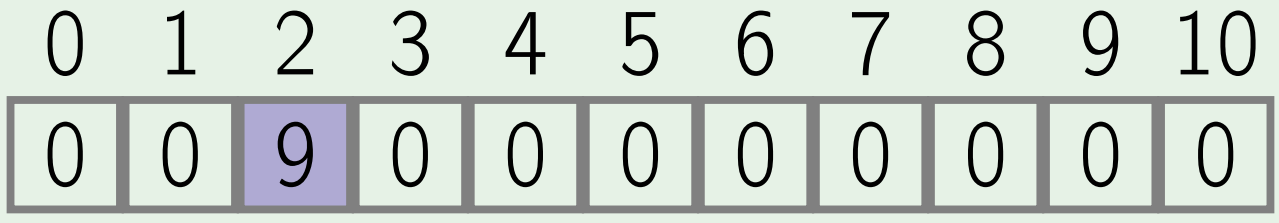
Example: $W = 10$



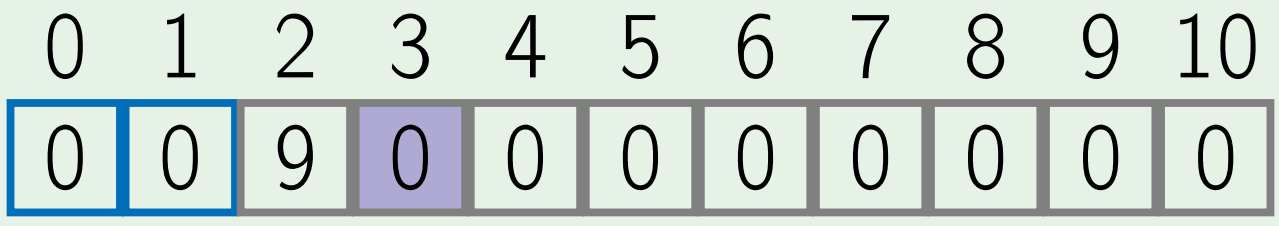
Example: $W = 10$



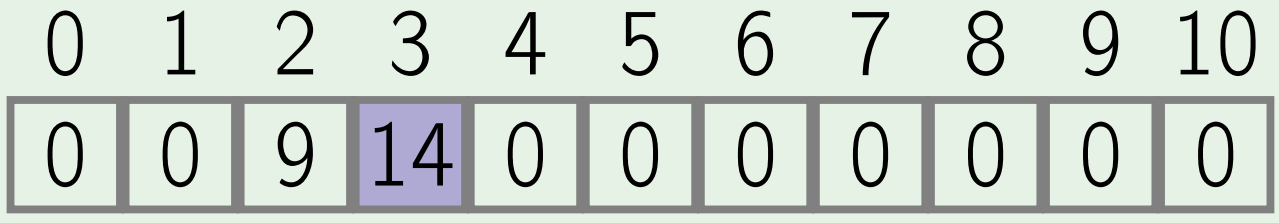
Example: $W = 10$



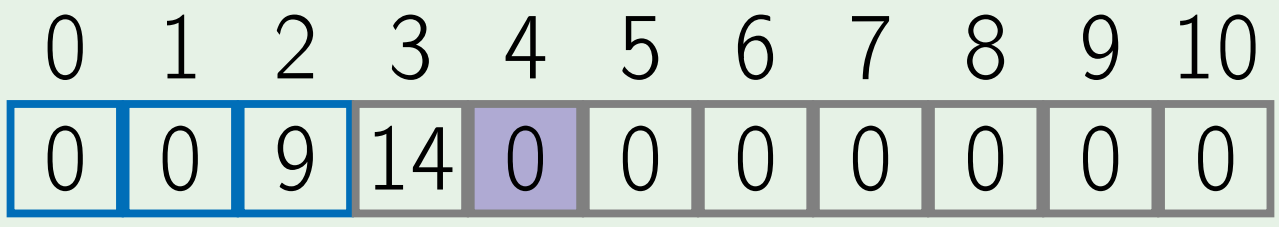
Example: $W = 10$



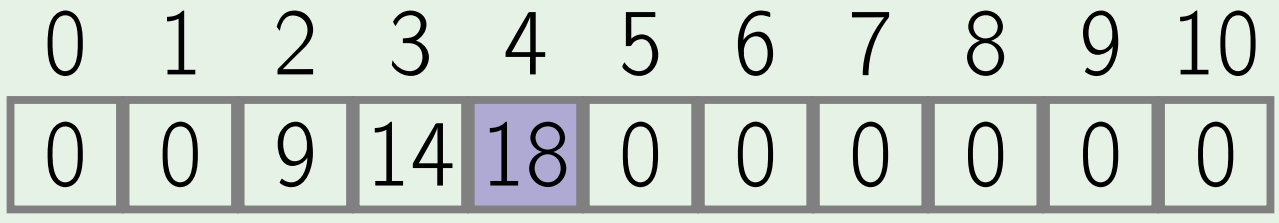
Example: $W = 10$



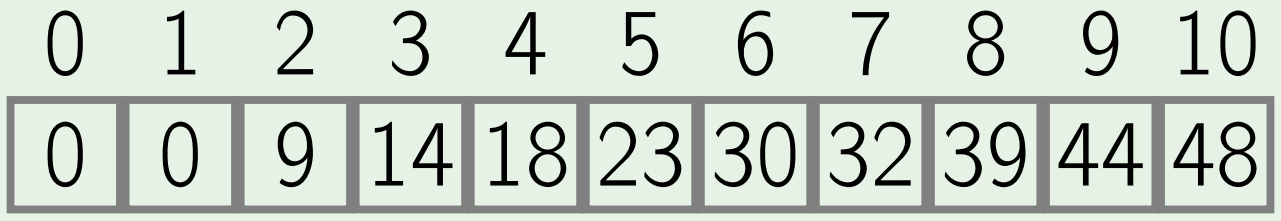
Example: $W = 10$



Example: $W = 10$



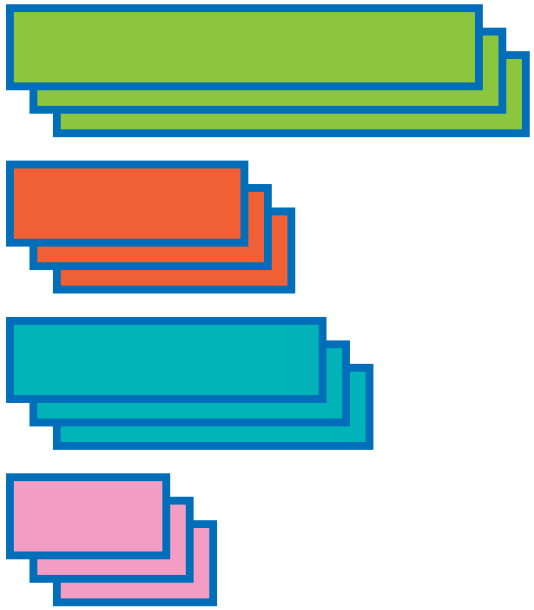
Example: $W = 10$



Outline

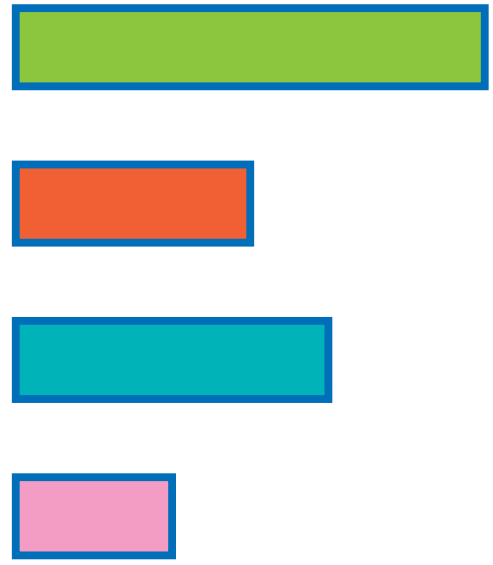
With repetitions:

unlimited quantities



Without repetitions:

one of each item

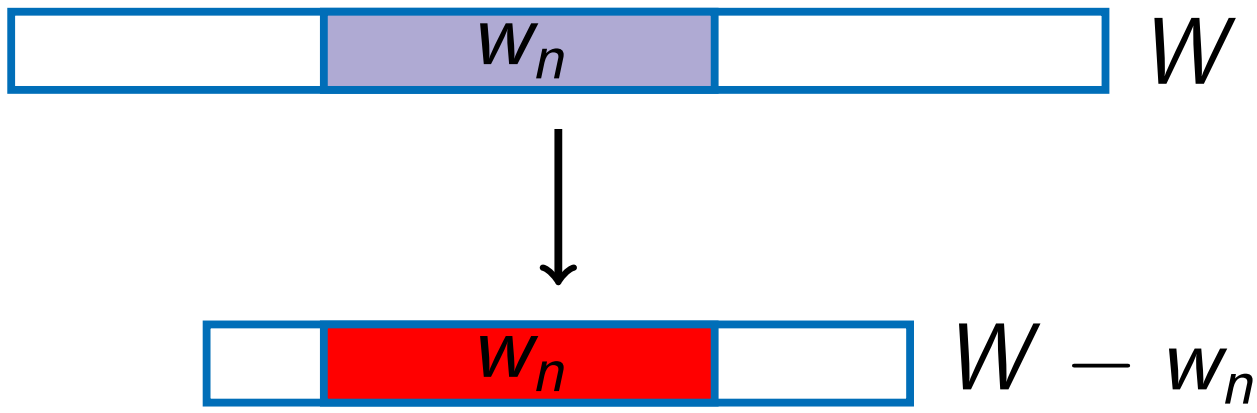


Knapsack without repetitions problem

Input: Weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ of n items; total weight W (v_i's, w_i's, and W are non-negative integers).

Output: The maximum value of items whose weight does not exceed W. Each item can be used at most once.

Same Subproblems?



Subproblems

- If the n-th item is taken into an optimal solution:

  then what is left is an optimal solution for a knapsack of total weight $W - w_n$ using items $1, 2, \ldots, n-1$.
- If the n-th item is not used, then the whole knapsack must be filled in optimally with items $1, 2, \ldots, n-1$.

Subproblems

For $0 \leq w \leq W$ and $0 \leq i \leq n$, $value(w, i)$ is the maximum value achievable using a knapsack of weight w and items $1, \ldots, i$.

The i-th item is either used or not:
$value(w, i)$ is equal to

$$\max\{value(w - w_i, i-1) + v_i, value(w, i-1)\}$$

Knapsack(W)

```
initialize all value(0, j) ← 0
initialize all value(w, 0) ← 0
for i from 1 to n:
  for w from 1 to W:
    value(w, i) ← value(w, i − 1)
    if w_i ≤ w:
      val ← value(w − w_i, i − 1) + v_i
      if value(w, i) < val
        value(w, i) ← val
return value(W, n)
```

Example: reconstructing a solution

$30	$14	$16	$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4

Example: reconstructing a solution

\$30	\$14	\$16	\$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4

Example: reconstructing a solution

\$30	\$14	\$16	\$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4
			0

Example: reconstructing a solution

$30	$14	$16	$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4
			0

Example: reconstructing a solution

\$30	\$14	\$16	\$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4
		1	0

Example: reconstructing a solution

$30	$14	$16	$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4
		1	0

Example: reconstructing a solution

\$30	\$14	\$16	\$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4
	0	1	0

Example: reconstructing a solution

\$30	\$14	\$16	\$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4
	0	1	0

Example: reconstructing a solution

$30	$14	$16	$9
6	3	4	2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Optimal solution:

1	2	3	4
1	0	1	0

Outline

Memoization

Knapsack(w)

```
if w is in hash table:
  return value(w)
value(w) ← 0
for i from 1 to n:
  if w_i ≤ w:
    val ← Knapsack(w − w_i) + v_i
    if val > value(w):
      value(w) ← val
insert value(w) into hash table with key w
return value(w)
```

What Is Faster?

- If all subproblems must be solved then an iterative algorithm is usually faster since it has no recursion overhead.
- There are cases however when one does not need to solve all subproblems: assume that W and all w_i's are multiples of 100; then $value(w)$ is not needed if w is not divisible by 100.

Running Time

- The running time $O(nW)$ is not polynomial since the input size is proportional to $\log W$, but not W.
- In other words, the running time is $O(n2^{\log W})$.
- E.g., for

$$W = 71\,345\,970\,345\,617\,824\,751$$

(twenty digits only!) the algorithm needs roughly 10^{20} basic operations.

Running Time

- The running time $O(nW)$ is not p
- I
- E

later, we'll learn why
solving this problem
in polynomial time costs \$1M!

(twenty digits only!) the algorithm needs roughly 10^{20} basic operations.

Dynamic Programming: Placing Parentheses

Alexander S. Kulikov

St. Petersburg Department of Steklov Institute of Mathematics
Russian Academy of Sciences

Data Structures and Algorithms
Algorithmic Toolbox

Outline

How to place parentheses in an expression

$$1 + 2 - 3 \times 4 - 5$$

to maximize its value?

Example

- $((((1 + 2) - 3) \times 4) - 5) = -5$
- $((1 + 2) - ((3 \times 4) - 5)) = -4$

Answer

$$((1 + 2) - (3 \times (4 - 5))) = 6$$

Another example

What about

$$5 - 8 + 7 \times 4 - 8 + 9\,?$$

Soon

We'll design an efficient dynamic programming algorithm to find the answer.

Outline

Placing parentheses

Input: A sequence of digits $d_1, \ldots, d_n$ and a sequence of operations $op_1, \ldots, op_{n-1} \in \{+, -, \times\}$.

Output: An order of applying these operations that maximizes the value of the expression

$$d_1\, op_1\, d_2\, op_2 \cdots op_{n-1}\, d_n\,.$$

Intuition

- Assume that the last operation in an optimal parenthesizing of $5 - 8 + 7 \times 4 - 8 + 9$ is $\times$:

$$(5 - 8 + 7) \times (4 - 8 + 9)\,.$$

- It would help to know optimal values for subexpressions $5 - 8 + 7$ and $4 - 8 + 9$.

However

We need to keep track for both the minimal and the maximal values of subexpressions!

Example: $(5 - 8 + 7) \times (4 - 8 + 9)$

$$\begin{aligned}
\min(5 - 8 + 7) = (5 - (8 + 7)) &= -10 \\
\max(5 - 8 + 7) = ((5 - 8) + 7) &= 4 \\
\min(4 - 8 + 9) = (4 - (8 + 9)) &= -13 \\
\max(4 - 8 + 9) = ((4 - 8) + 9) &= 5
\end{aligned}$$

$$\max((5 - 8 + 7) \times (4 - 8 + 9)) = 130$$

Subproblems

- Let $E_{i,j}$ be the subexpression

$$d_i \, op_i \cdots op_{j-1} \, d_j$$

- Subproblems:

$$M(i,j) = \text{maximum value of } E_{i,j}$$
$$m(i,j) = \text{minimum value of } E_{i,j}$$

Recurrence Relation

$$M(i,j) = \max_{i \le k \le j-1} \begin{cases} M(i,k) & op_k & M(k+1,j) \\ M(i,k) & op_k & m(k+1,j) \\ m(i,k) & op_k & M(k+1,j) \\ m(i,k) & op_k & m(k+1,j) \end{cases}$$

$$m(i,j) = \min_{i \le k \le j-1} \begin{cases} M(i,k) & op_k & M(k+1,j) \\ M(i,k) & op_k & m(k+1,j) \\ m(i,k) & op_k & M(k+1,j) \\ m(i,k) & op_k & m(k+1,j) \end{cases}$$

Outline

MinAndMax(i, j)

$min \leftarrow +\infty$

$max \leftarrow -\infty$

for k from i to $j-1$:

$a \leftarrow M(i,k) \quad op_k \quad M(k+1,j)$

$b \leftarrow M(i,k) \quad op_k \quad m(k+1,j)$

$c \leftarrow m(i,k) \quad op_k \quad M(k+1,j)$

$d \leftarrow m(i,k) \quad op_k \quad m(k+1,j)$

$min \leftarrow \min(min, a, b, c, d)$

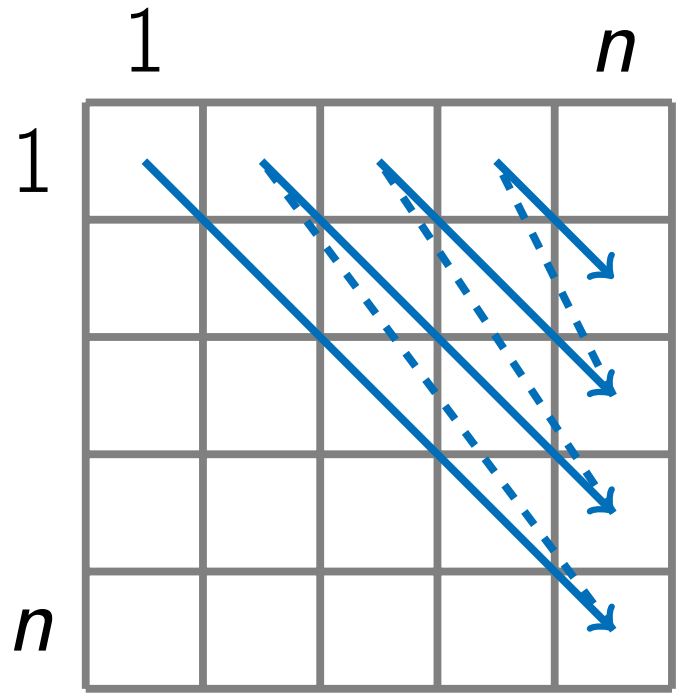
$max \leftarrow \max(max, a, b, c, d)$

return (min, max)

Order of Subproblems

- When computing $M(i, j)$, the values of $M(i, k)$ and $M(k + 1, j)$ should be already computed.
- Solve all subproblems in order of increasing $(j - i)$.

Possible Order



Parentheses($d_1\ op_1\ d_2\ op_2 \ldots d_n$)

```
for i from 1 to n:
    m(i,i) ← d_i ,  M(i,i) ← d_i
for s from 1 to n − 1:
    for i from 1 to n − s:
        j ← i + s
        m(i,j), M(i,j) ← MinAndMax(i,j)
return M(1,n)
```

O(n^3)

Example: $5 - 8 + 7 \times 4 - 8 + 9$

5	-3	-10	-55	-63	-94
	8	15	36	-60	-195
		7	28	-28	-91
			4	-4	-13
				8	17
					9

m

5	-3	4	25	65	200
	8	15	60	52	75
		7	28	20	35
			4	-4	5
				8	17
					9

M

Outline

Example: $5 - 8 + 7 \times 4 - 8 + 9$

5	-3	-10	-55	-63	-94
	8	15	36	-60	-195
		7	28	-28	-91
			4	-4	-13
				8	17
					9

m

5	-3	4	25	65	200
	8	15	60	52	75
		7	28	20	35
			4	-4	5
				8	17
					9

M