Algorithmics and Complexity
Cours 5/7 : Dynamic Programming

CentraleSupélec – Gif

ST2 – Gif

Plan

Change making

Change making

Give back 3,57 EUR with coins worth 1 and 2 euros and 1, 2, 5, 10, 20 and 50 cents.

Change making

Change making

Give back 3,57 EUR with coins worth 1 and 2 euros and 1, 2, 5, 10, 20 and 50 cents.

Solutions

A set of coins:

➜ 2€ + 1€ + 50¢ + 5¢ + 2¢

Change making

Change making

Give back 3,57 EUR with coins worth 1 and 2 euros and 1, 2, 5, 10, 20 and 50 cents.

Solutions

A set of coins:

- ✓ 2€ + 1€ + 50¢ + 5¢ + 2¢
- ➜ 1€ + 1€ + 50¢ + 4×20¢ + 2×10¢ + 3×2¢ + 1¢

Change making

Change making

Give back 3,57 EUR with coins worth 1 and 2 euros and 1, 2, 5, 10, 20 and 50 cents.

Solutions

A set of coins:

- ✓ 2€ + 1€ + 50¢ + 5¢ + 2¢
- ✓ 1€ + 1€ + 50¢ + 4×20¢ + 2×10¢ + 3×2¢ + 1¢

Change making

Change making

Give back 3,57 EUR with coins worth 1 and 2 euros and 1, 2, 5, 10, 20 and 50 cents.

Solutions

A set of coins:

- ✓ 2€ + 1€ + 50¢ + 5¢ + 2¢
- ✓ 1€ + 1€ + 50¢ + 4×20¢ + 2×10¢ + 3×2¢ + 1¢
- ✓ 357×1¢

Change making

Change making

Give back 3,57 EUR with coins worth 1 and 2 euros and 1, 2, 5, 10, 20 and 50 cents.

Solutions

A set of coins:

- ✓ 2€ + 1€ + 50¢ + 5¢ + 2¢
- ✓ 1€ + 1€ + 50¢ + 4×20¢ + 2×10¢ + 3×2¢ + 1¢
- ✓ 357×1¢
- ➜ With how many minimum number of coins can we return 3,57€?

Optimization problem

Input data

- $S \in \mathbb{N}^{+n}$ a n-tuple of coins: $S = (200, 100, 50, 20, 10, 5, 2, 1)$
- $total \in \mathbb{N}$ the amount to give back: $total = 357$

Output data

c the number of coins used to obtain the value $total$,
as it exists $L \in \mathbb{N}^n$ a n-tuple, checking:

- $total = \sum_{i=0}^{n-1} L_i \times S_i$ *L indicates the number of each piece*

Optimization problem

Input data

- $S \in \mathbb{N}^{+n}$ a n-tuple of coins: $S = (200, 100, 50, 20, 10, 5, 2, 1)$
- $total \in \mathbb{N}$ the amount to give back: $total = 357$

Output data

c the number of coins used to obtain the value $total$,
as it exists $L \in \mathbb{N}^n$ a n-tuple, checking:

- $total = \sum_{i=0}^{n-1} L_i \times S_i$ *L indicates the number of each piece*
- $c = \sum_{i=0}^{n-1} L_i$ *c is the cost of the solution L*

Optimization problem

Input data

- $S \in \mathbb{N}^{+n}$ a n-tuple of coins: $S = (200, 100, 50, 20, 10, 5, 2, 1)$
- $total \in \mathbb{N}$ the amount to give back: $total = 357$

Output data

c the number of coins used to obtain the value $total$,
as it exists $L \in \mathbb{N}^n$ a n-tuple, checking:

- $total = \sum_{i=0}^{n-1} L_i \times S_i$ *L indicates the number of each piece*
- $c = \sum_{i=0}^{n-1} L_i$ *c is the cost of the solution L*
- and c is minimal!

Example

$L = (0, 2, 1, 4, 2, 0, 3, 1) \rightarrow c = 13$ ➜ not minimal!

Some observations. . .

Solution

Coins of unit value $\rightarrow$ at least one solution $\forall total \in \mathbf{N}$

We assume an infinity of coins for each of the values. . .

Problem size ?

Some observations. . .

Solution

Coins of unit value $\rightarrow$ at least one solution $\forall total \in \mathbf{N}$

We assume an infinity of coins for each of the values. . .

Problem size

n (the number of different coin values)

➜ The sum to be returned is a **parameter** of the problem

Some observations. . .

Solution

Coins of unit value $\rightarrow$ at least one solution $\forall total \in \mathbf{N}$

We assume an infinity of coins for each of the values. . .

Problem size

n (the number of different coin values)

➜ The sum to be returned is a **parameter** of the problem

Goal

- ✓ Return the sum $\rightarrow$ a solution
- ✓ With a minimum number of coins $\rightarrow$ the optimal solution!

Some observations...

Solution

Coins of unit value → at least one solution $\forall total \in \mathbf{N}$

We assume an infinity of coins for each of the values...

Problem size

n (the number of different coin values)

➜ The sum to be returned is a **parameter** of the problem

Goal

- ✓ Return the sum → a solution
- ✓ With a minimum number of coins → the optimal solution!

Optimal solution

We are looking for the cost of the optimal solution!

Recursive algorithm

To return 3,57€, I can return:

- one coin of 2 € then 1,57;
- one coin of 1 € then 2,57;
- one coin of 50 ¢ then 3,07;
- *etc.* *for every possible coin value*

Recursive algorithm

To return 3,57€, I can return:

- one coin of 2 € then 1,57;
- one coin of 1 € then 2,57;
- one coin of 50 ¢ then 3,07;
- *etc.* *for every possible coin value*

➜ The best solution is then:

$$1 + min(given\_back(2, 57), given\_back(1, 57), \ldots)$$

Recursive algorithm

To return 3,57€, I can return:

- one coin of 2 € then 1,57;
- one coin of 1 € then 2,57;
- one coin of 50 ¢ then 3,07;
- *etc. for every possible coin value*

➜ The best solution is then:

$$1 + min(given\_back(2,57), given\_back(1,57), \ldots)$$

Recursive calculation of the cost of the optimal solution

Denote by $C(s)$ the minimum number of coins to obtain s.

Recursive algorithm

To return 3,57€, I can return:

- one coin of 2 € then 1,57;
- one coin of 1 € then 2,57;
- one coin of 50 ¢ then 3,07;
- *etc.* *for every possible coin value*

➜ The best solution is then:

$$1 + min(given\_back(2,57), given\_back(1,57), \ldots)$$

Recursive calculation of the cost of the optimal solution

Denote by $C(s)$ the minimum number of coins to obtain s.

- Base case: $C(0) = 0$

Recursive algorithm

To return 3,57€, I can return:

- one coin of 2 € then 1,57;
- one coin of 1 € then 2,57;
- one coin of 50 ¢ then 3,07;
- *etc.* *for every possible coin value*

➜ The best solution is then:

$$1 + min(given\_back(2,57), given\_back(1,57), \ldots)$$

Recursive calculation of the cost of the optimal solution

Denote by $C(s)$ the minimum number of coins to obtain s.

- Base case: $C(0) = 0$
- General case: $C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i)$

Recursive approach

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0, n-1], S_i \leq s} C(s - S_i) \end{cases}$$

Recursive approach

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

357: $(0, 0, 0, 0, 0, 0, 0, 0)$

Recursive approach

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

357: $(0, 0, 0, 0, 0, 0, 0, 0)$

157: $(1, 0, 0, 0, 0, 0, 0, 0)$

Recursive approach

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0, n-1], S_i \leq s} C(s - S_i) \end{cases}$$

357: $(0, 0, 0, 0, 0, 0, 0, 0)$

157: $(1, 0, 0, 0, 0, 0, 0, 0)$ **257:** $(0, 1, 0, 0, 0, 0, 0, 0)$

Recursive approach

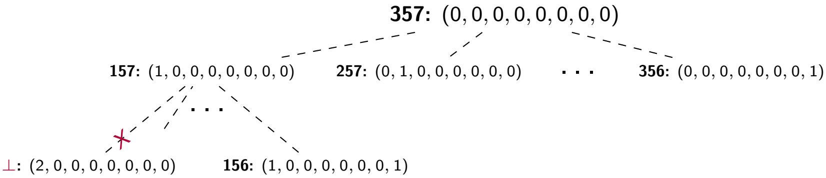
$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

357: $(0,0,0,0,0,0,0,0)$

157: $(1,0,0,0,0,0,0,0)$ **257:** $(0,1,0,0,0,0,0,0)$ . . . **356:** $(0,0,0,0,0,0,0,1)$

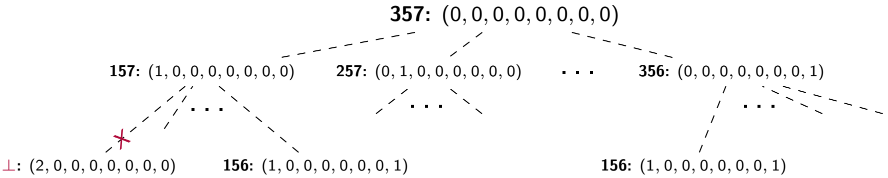
Recursive approach

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



Recursive approach

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



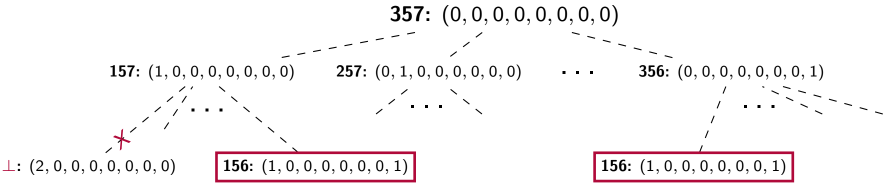
Exponential complexity!

- Unbalanced n-ary exploration tree of depth within $[\frac{total}{S_0}, \frac{total}{S_{n-1}}]$
- Complexity higher than n^k where $k = \frac{total}{S_0}$.

mais. . .

Recursive approach

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



Exponential complexity!

- Unbalanced n-ary exploration tree of depth within $[\frac{total}{S_0}, \frac{total}{S_{n-1}}]$
- Complexity higher than n^k where $k = \frac{total}{S_0}$.
- Redundant computation!

Dynamic programming

Idea

- ✓ Solve optimization problems
- ✓ Where there is a recursive construction of the solution

Dynamic programming

Idea

- ✓ Solve optimization problems
- ✓ Where there is a recursive construction of the solution
- ➜ **Dynamic programming**

Principle

- Store the intermediate solutions so as not to recalculate them
- Invented by Bellman in the 1950s
- Applies when the optimal solution of the problem is composed of the optimal solutions of its subproblems

Plan

Dynamic programming

Principle

- Recursively, start by solving the smallest sub-problems, then solve bigger and bigger sub-problems until the solution to the global problem is obtained.

Dynamic programming

Principle

- Recursively, start by solving the smallest sub-problems, then solve bigger and bigger sub-problems until the solution to the global problem is obtained.
- Keep the solutions to sub-problems in a table

  to avoid redundant computations that make the recursive solution inefficient

Dynamic programming

Principle

- Recursively, start by solving the smallest sub-problems, then solve bigger and bigger sub-problems until the solution to the global problem is obtained.
- Keep the solutions to sub-problems in a table

  to avoid redundant computations that make the recursive solution inefficient

Example: change making

We iterate from $s = 0$ to $s = total$

Dynamic programming

Principle

- Recursively, start by solving the smallest sub-problems, then solve bigger and bigger sub-problems until the solution to the global problem is obtained.
- Keep the solutions to sub-problems in a table

  to avoid redundant computations that make the recursive solution inefficient

Example: change making

We iterate from $s = 0$ to $s = total$

- As we know how to make change for all $s' < s$:

  $\rightarrow$ we compute the min cost for s:

  $C(s) = 1 + \min_{i \in [0, n-1], S_i \leq s} C(s - S_i)$

Dynamic programming

Principle

- Recursively, start by solving the smallest sub-problems, then solve bigger and bigger sub-problems until the solution to the global problem is obtained.
- Keep the solutions to sub-problems in a table

  to avoid redundant computations that make the recursive solution inefficient

Example: change making

We iterate from $s = 0$ to $s = total$

- As we know how to make change for all $s' < s$:

  $\rightarrow$ we compute the min cost for s:

  $C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i)$
- We memorize the result in the array

  $\rightarrow$ needed later to compute the min cost of $s + S_i$

Dynamic programming

Optimal sub-structure

1. Divide the problem in sub-problems
2. Construct the optimal solution from optimal solutions of sub-problems
3. Deduce a recurrence formula

Dynamic programming

Optimal sub-structure

1. Divide the problem in sub-problems
2. Construct the optimal solution from optimal solutions of sub-problems
3. Deduce a recurrence formula

Examples of applications

- Sequence alignment
- Change making
- Shortest path
- Knapsack

Dynamic programming vs Divide and conquer

Similarity

Both methods need an optimal sub-structure (a recurrence formula)

Dynamic programming vs Divide and conquer

Similarity

Both methods need an optimal sub-structure (a recurrence formula)

- If the sub-problems are independents (all the sub-problems are different)
  - Dynamic programming is useless
  - classic example: $fact(n+1) = (n+1) \times fact(n)$

Dynamic programming vs Divide and conquer

Similarity

Both methods need an optimal sub-structure (a recurrence formula)

- If the sub-problems are independents (all the sub-problems are different)
  - Dynamic programming is useless
  - classic example: $fact(n+1) = (n+1) \times fact(n)$
- Otherwise
  - Dynamic programming is more efficient in time
    (in return, we pay in space because nothing is free!)
  - classic example: $fib(n+2) = fib(n+1) + fib(n)$

Dynamic programming vs Divide and conquer

- Divide and conquer (recursive approach)

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

  - exponential complexity $\mathcal{O}(\phi^n)$ (ϕ the golden ratio)

Dynamic programming vs Divide and conquer

- Divide and conquer (recursive approach)

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

  - exponential complexity $\mathcal{O}(\phi^n)$ (ϕ the golden ratio)

- Dynamic programming

```
table = {0:0, 1:1}
def fib(n):
  if not n in table:
    table[n] = fib(n-1) + fib(n-2)
  return table[n]
```

  - linear complexity $\mathcal{O}(n)$

Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 \quad \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0, n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1			1					1				

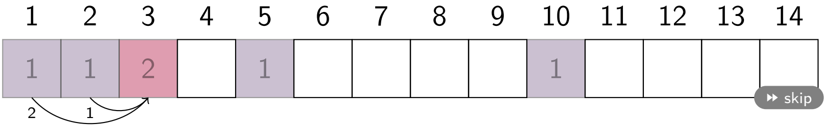
⏩ skip

Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 \quad \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0, n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

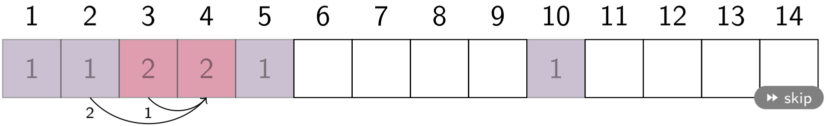


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

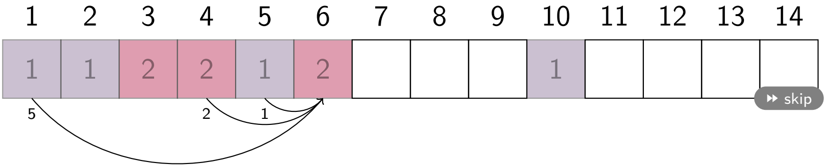


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

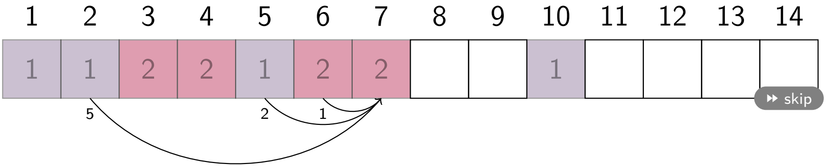


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 \quad \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

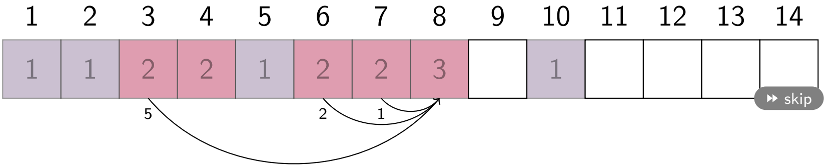


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

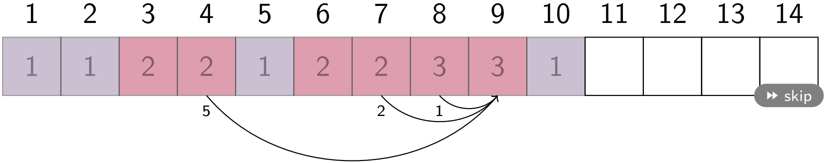


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

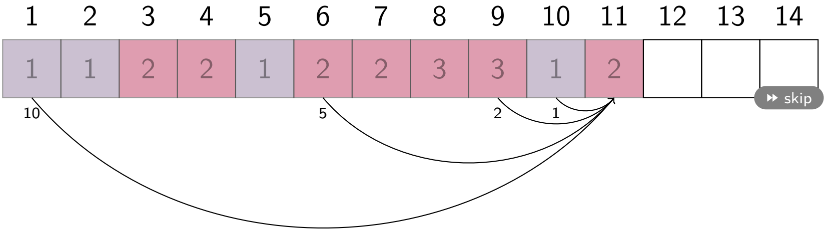


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

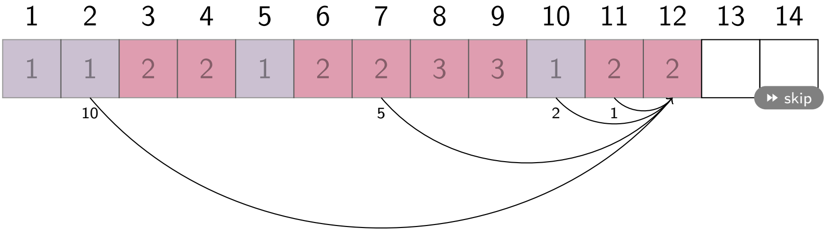


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

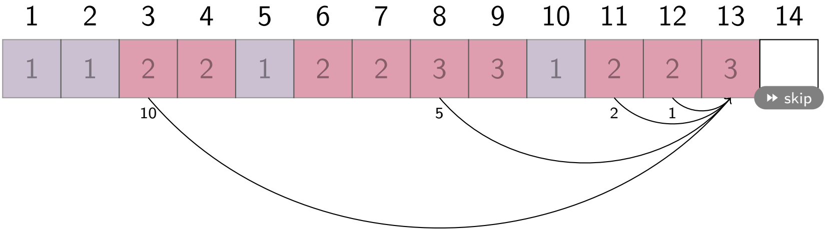


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 \quad \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$

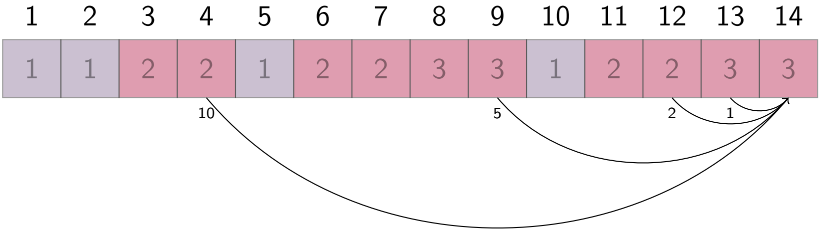


Resolution with dynamic programming (Algorithm 1)

- We reuse the previous recurrence formula while saving the intermediate results:

$$\begin{cases} C(s) = 1 \quad \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0, n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Let $S = (10, 5, 2, 1)$ and $total = 14$



- computation time $n \times total$

Algorithm 1, recursive version

```
import math
S = (200, 100, 50, 20, 10, 5, 2, 1); n=len(S)
total = 357;
C = [math.inf for i in range(total+1)]

def given_back(sum):
    if C[sum]==math.inf:
        if sum in S:
            C[sum]=1
        else:
            best = math.inf
            for i in range(n):
                if S[i]<sum:
                    best = min(best,given_back(sum-S[i]))
            C[sum] = best+1
    return C[sum]

print("Number_of_coins:", given_back(total))
```

Algorithm 1, iterative version

```
import math
S = (200, 100, 50, 20, 10, 5, 2, 1); n=len(S)
total = 357;
C = [0]

for i in range(1,total+1):
    C.append(math.inf)
    for j in range(n):
        if i>=S[j] and 1+C[i-S[j]]<C[i]:
            C[i] = 1+C[i-S[j]]

print("Number_of_coins:", str(C[total]))
```

Algorithm 1, iterative version

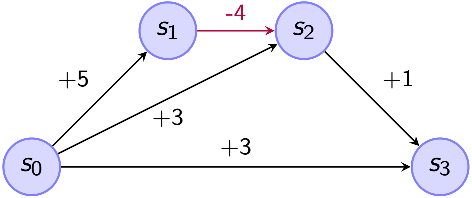
```
import math
S = (200, 100, 50, 20, 10, 5, 2, 1); n=len(S)
total = 357;
C = [0]

for i in range(1,total+1):
    C.append(math.inf)
    for j in range(n):
        if i>=S[j] and 1+C[i-S[j]]<C[i]:
            C[i] = 1+C[i-S[j]]

print("Number_of_coins:", str(C[total]))
```

➜ In both cases, we do not have the solution, only its cost $C(total)$.

Plan

1. Change making
2. Dynamic programming
3. Shortest Path
   - Shortest Paths algorithm
   - Bellman-Ford
   - Algorithm
   - Negative weight cycles detection
   - Application: routing
4. Conclusion
5. Sequence alignment

The Shortest Paths algorithm does not work with negative weights

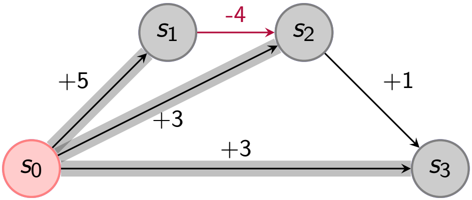


Node	Distance	Parent
s_0	0	•
s_1	∞	•
s_2	∞	•
s_3	∞	•

Frontier = $\{s_0\}$

x =

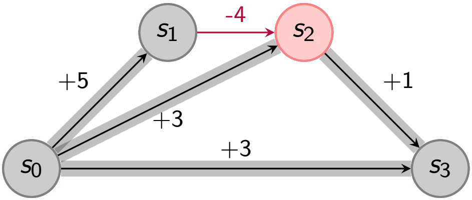
The Shortest Paths algorithm does not work with negative weights



Node	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	3	s_0
s_3	3	s_0

$$\text{Frontier} = \{s_1, s_2, s_3\}$$
$$x = s_0$$

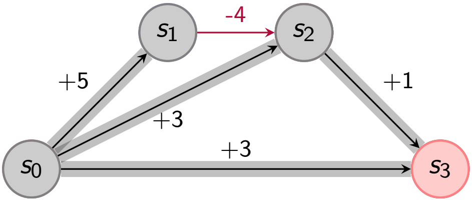
The Shortest Paths algorithm does not work with negative weights



Node	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	3	s_0
s_3	3	s_0

Frontier $= \{s_1, s_3\}$
$x = s_2$

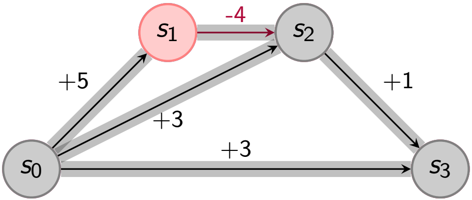
The Shortest Paths algorithm does not work with negative weights



Node	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	3	s_0
s_3	3	s_0

Frontier $= \{s_1\}$
$x = s_3$

The Shortest Paths algorithm does not work with negative weights

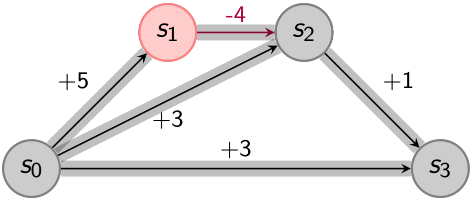


Node	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	1	s_1
s_3	3	s_0

Frontier =

x = s_1

The Shortest Paths algorithm does not work with negative weights



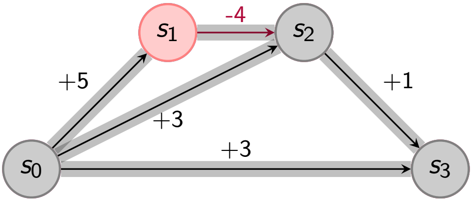
Node	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	1	s_1
s_3	3	s_0

Frontier =

x =

The Shortest Paths algorithm gives a wrong answer for s_3!

The Shortest Paths algorithm does not work with negative weights



Node	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	1	s_1
s_3	3	s_0

Frontier =

x =

The Shortest Paths algorithm gives a wrong answer for s_3!

Why would someone want to calculate a shortest path on a graph with negative weights?

➜ answer to TD 4, exercise 1 (placement problem)

Bellman-Ford algorithm (1956, 1958)

Principle

- Based on the principle of the dynamic programming
- Calculate the cost of the shortest path
  but we can recover the path from the memoization table...

Bellman-Ford algorithm (1956, 1958)

Principle

- Based on the principle of the dynamic programming
- Calculate the cost of the shortest path

  but we can recover the path from the memoization table...

Reminder: problem data

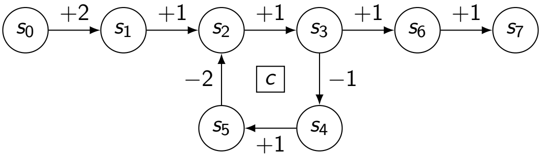
An arbitrary weighted and directed graph G, two vertices s and t

including negative weights...

➜ What is the length of the shortest path from s to t?

Attention to the negative weight cycles!

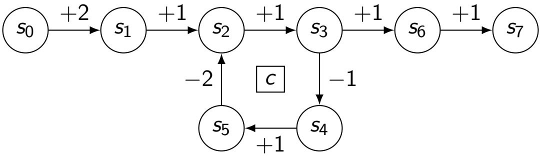
Definition: negative weight cycle



The cycle c in this example is a *negative weight cycle*, because $\sum_{e=(v,u)\in c} \omega(e) < 0$.

Attention to the negative weight cycles!

Definition: negative weight cycle



The cycle c in this example is a *negative weight cycle*, because $\sum_{e=(v,u)\in c} \omega(e) < 0$.

Shortest path with negative weights

Need a more precise formulation:

➜ We are looking for the shortest path without cycle!

Bellman-Ford algorithm (1956, 1958)

Principle

- Based on the principle of the dynamic programming
- Calculate the cost of the shortest path
  but we can recover the path from the memoization table...

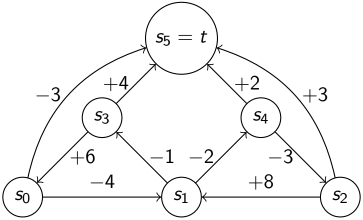
Properties

- ✓ Supports negative weights (unlike Dijkstra)
- ✓ Detects if there is a negative weight cycle

Principles of the Bellman-Ford algorithm

Divide into subproblems

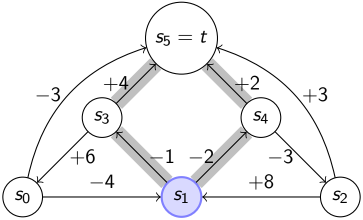
Let $\mathrm{OPT}(i, v)$ be the length of the shortest path to the target node t from a node $v, v \neq t$, which contains at most i arcs.



Principles of the Bellman-Ford algorithm

Divide into subproblems

Let OPT(i, v) be the length of the shortest path to the target node t from a node v, $v \neq t$, which contains at most i arcs.

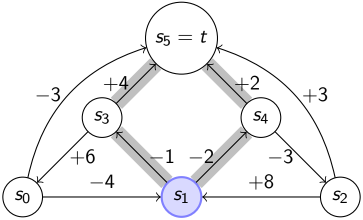


OPT(2,s1)=?

Principles of the Bellman-Ford algorithm

Divide into subproblems

Let OPT(i, v) be the length of the shortest path to the target node t from a node v, $v \neq t$, which contains at most i arcs.



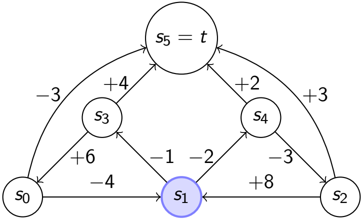
OPT(2,s1)=?

the length of the shortest path in 2 arcs from s1 to t (s5)
(in this case, it is 0 through s4)

Principles of the Bellman-Ford algorithm

Divide into subproblems

Let OPT(i, v) be the length of the shortest path to the target node t from a node v, $v \neq t$, which contains at most i arcs.

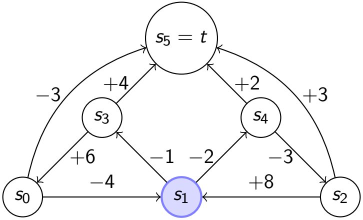


OPT($|V| - 1$,s1)=?

Principles of the Bellman-Ford algorithm

Divide into subproblems

Let $\mathrm{OPT}(i, v)$ be the length of the shortest path to the target node t from a node $v, v \neq t$, which contains at most i arcs.



$\mathrm{OPT}(|V| - 1, s1) = ?$

the length of the shortest path from s1 to t (s5)
(in this case, it is -2)

Principles of the Bellman-Ford algorithm

Divide into subproblems

Let $\mathrm{OPT}(i, v)$ be the length of the shortest path to the target node t from a node $v, v \neq t$, which contains at most i arcs.

Recursive construction of a solution

- $\mathrm{OPT}(i, v) = \min_{(v,u)\in E}\big(\mathrm{OPT}(i-1, u) + \omega((v, u))\big)$

  $\rightarrow$ To reach t, first go to u by taking the shortest path in $i-1$ steps.

Principles of the Bellman-Ford algorithm

Divide into subproblems

Let $\text{OPT}(i, v)$ be the length of the shortest path to the target node t from a node v, $v \neq t$, which contains at most i arcs.

Recursive construction of a solution

- $\text{OPT}(i, v) = \min_{(v,u)\in E}\big(\text{OPT}(i-1, u) + \omega((v, u))\big)$

  $\rightarrow$ To reach t, first go to u by taking the shortest path in $i-1$ steps.
- Unless there is already a path in $i-1$ steps from v which is shorter than all the rest!

  $\rightarrow$ In which case $\text{OPT}(i, v) = \text{OPT}(i-1, v)$

Principles of the Bellman-Ford algorithm

Divide into subproblems

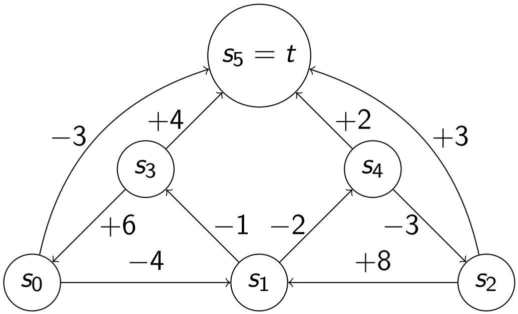
Let $\mathrm{OPT}(i, v)$ be the length of the shortest path to the target node t from a node v, $v \neq t$, which contains at most i arcs.

Recursive construction of a solution

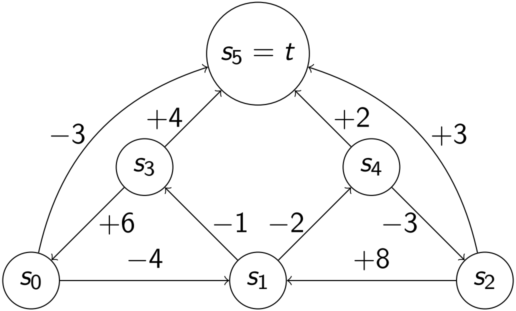
$$\mathrm{OPT}(i, v) = \min\left(\mathrm{OPT}(i-1, v), \min_{u \in V}(\mathrm{OPT}(i-1, u) + \omega((v, u)))\right)$$

Store the $\mathrm{OPT}(i, v) \rightarrow$ 2-dimensional array.

Example

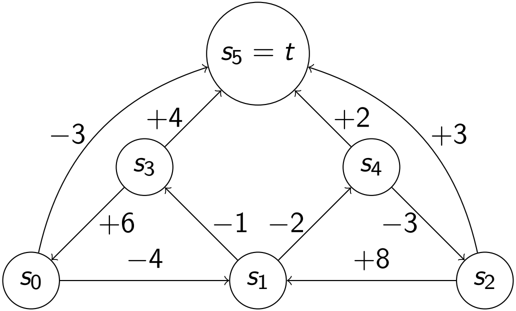


Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0

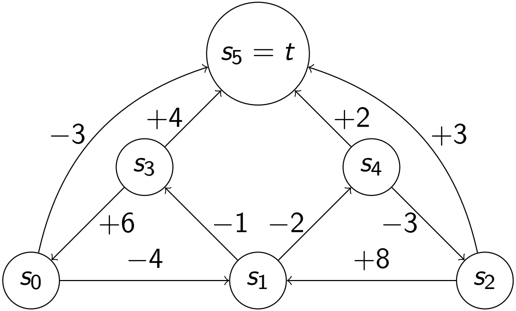
Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3					

$$s_0 = min(\infty, min(-4 + \infty(by\ s_1), -3 + 0(by\ s_5)))$$

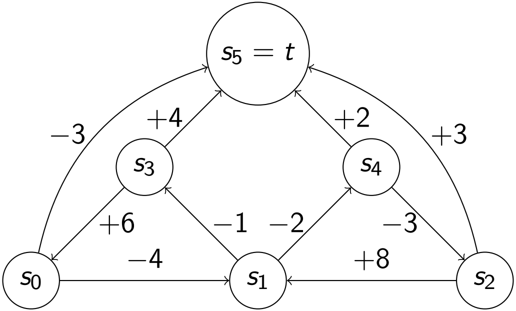
Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞				

$$s_1 = min(\infty, min(-1 + \infty, -2 + \infty))$$

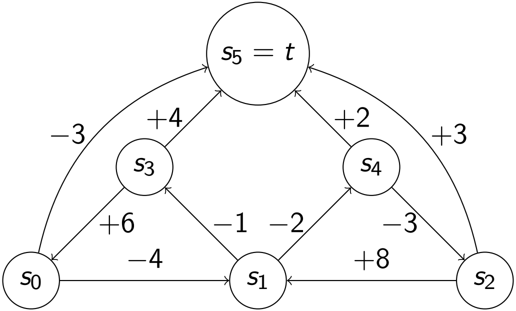
Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3			

$$s_2 = min(\infty, min(8 + \infty, 3 + 0))$$

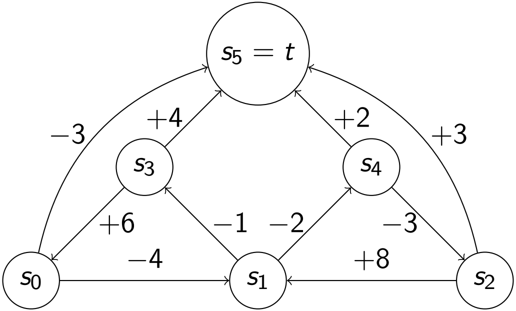
Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0

we end the line on the same principle

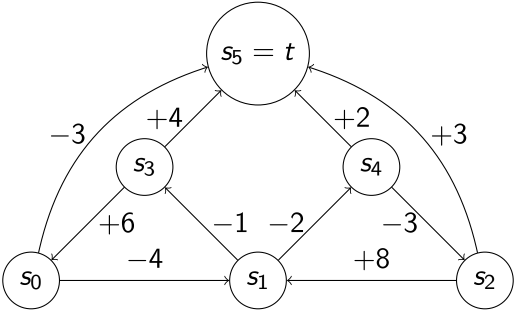
Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0

then we do the next line

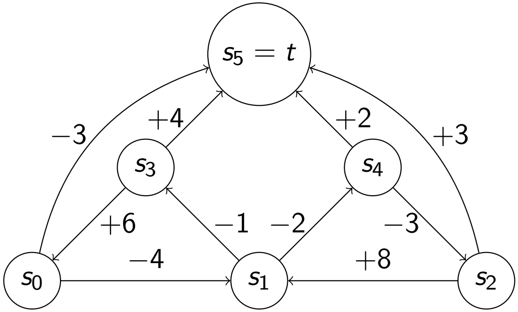
Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0

and so on...

Example



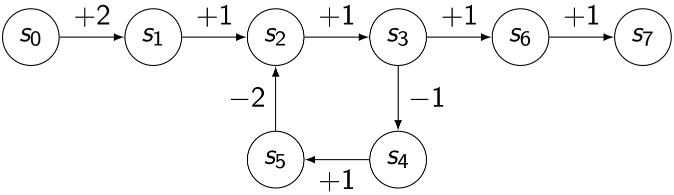
	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0
4	-6	-2	3	2	0	0
5	-6	-2	3	0	0	0

When to stop?

Proprerty

- In an acyclic graph, the shortest path contains at most $|V| - 1$ arcs

➜ This is also true in every graph without negative weight cycle

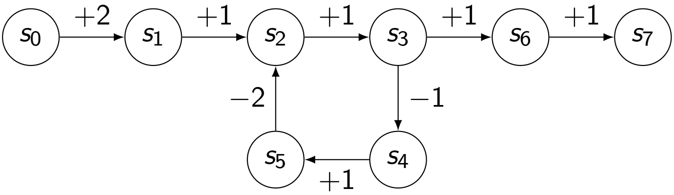


When to stop?

Proprerty

- In an acyclic graph, the shortest path contains at most $|V| - 1$ arcs

➜ This is also true in every graph without negative weight cycle



➜ Stop as soon as you have computed the paths with $|V| - 1$ arcs.

One implementation of Bellman-Ford (adjacency matrix)

```
import math

# graph is an adjacency matrix
n = len(graph)

# initialization of OPT table
OPT = [[math.inf for _ in range(n)] for _ in range(n)]
OPT[0][5] = 0

# filling the table
for i in range(1,n):
    for v in range(n):
        OPT[i][v] = OPT[i-1][v]
        for u in range(n):
            if graph[v][u] != None and \
                    OPT[i][v] > OPT[i-1][u] + graph[v][u]:
                OPT[i][v] = OPT[i-1][u] + graph[v][u]
```

Bellman-Ford complexity

Complexity **adjacency matrix**

- 3 nested loops of $|V|$ iterations each
- an access in $\mathcal{O}(1)$ to $\omega((v,u))$ at each turn of the inner loop!

Bellman-Ford complexity

Complexity **adjacency matrix**

- 3 nested loops of $|V|$ iterations each
- an access in $\mathcal{O}(1)$ to $\omega((v, u))$ at each turn of the inner loop!

➜ Hence, the total complexity of the algorithm is: $\mathcal{O}(|V|^3)$.

Bellman-Ford complexity

Complexity — adjacency matrix

- 3 nested loops of $|V|$ iterations each
- an access in $\mathcal{O}(1)$ to $\omega((v, u))$ at each turn of the inner loop!

➜ Hence, the total complexity of the algorithm is: $\mathcal{O}(|V|^3)$.

Complexity — adjacency list

- do it home.
- 2 loops: for each of the $|V|$ lines, we iterate over the $|E|$ edges

Bellman-Ford complexity

Complexity **adjacency matrix**

- 3 nested loops of $|V|$ iterations each
- an access in $\mathcal{O}(1)$ to $\omega((v,u))$ at each turn of the inner loop!

➜ Hence, the total complexity of the algorithm is: $\mathcal{O}(|V|^3)$.

Complexity **adjacency list**

- do it home.
- 2 loops: for each of the $|V|$ lines, we iterate over the $|E|$ edges

➜ Hence, the total complexity of the algorithm is: $\mathcal{O}(|V| \times |E|)$.

Negative weight cycles detection ($\Longrightarrow$)

Proprerty (reminder)

- In an graph without negative cycle, the shortest path has at most $|V| - 1$ arcs.

➜ $\forall v, \mathrm{OPT}(|V| - 1, v)$ is the length of the shortest path

Negative weight cycles detection ($\Longrightarrow$)

Proprerty (reminder)

- In an graph without negative cycle, the shortest path has at most $|V| - 1$ arcs.

➜ $\forall v, \text{OPT}(|V| - 1, v)$ is the length of the shortest path

Contraposition

If a shortest path holds more than $|V| - 1$ arcs, then G has negative cycles.

Negative weight cycles detection ($\Longrightarrow$)

Proprerty (reminder)

- In an graph without negative cycle, the shortest path has at most $|V| - 1$ arcs.

➜ $\forall v, \mathrm{OPT}(|V| - 1, v)$ is the length of the shortest path

Contraposition

If a shortest path holds more than $|V| - 1$ arcs, then G has negative cycles.

Corollary 1 ($\Longrightarrow$)

If one value in the row $|V|$ is smaller than the one of the previous row:

$$\exists v.\ \ OPT(|V|, v) < OPT(|V| - 1, v)$$

then there is a negative cycle in the graph.

Negative weight cycles detection $(\Longleftarrow)$

Properties

1. If G contains a negative cycle, then for any node v from that cycle, we can always improve its distance.
   - ➜ $\exists v.\ \forall n.\ \exists m > n \ \ OPT(m, v) < OPT(n, v)$

Negative weight cycles detection ($\Longleftarrow$)

Properties

1. If G contains a negative cycle, then for any node v from that cycle, we can always improve its distance.

   ➜ $\exists v.\ \ \forall n.\ \ \exists m > n\ \ OPT(m, v) < OPT(n, v)$

2. If a row of the table is equal to the next one, then all the following rows are equal to it as well
   - consequence of the recurrence formula

Negative weight cycles detection

$(\Longleftarrow)$

Properties

1. If G contains a negative cycle, then for any node v from that cycle, we can always improve its distance.

   ➜ $\exists v.\ \forall n.\ \exists m > n\ \ OPT(m, v) < OPT(n, v)$

2. If a row of the table is equal to the next one, then all the following rows are equal to it as well
   - consequence of the recurrence formula

Corollary 2

$(\Longleftarrow)$

If G contains a negative cycle, **then**:

$$\exists v.\ \ OPT(|V|, v) < OPT(|V| - 1, v)$$

Negative weight cycles detection

Theorem

G contains a negative cycle **iff**

$$\exists v.\ \ OPT(|V|, v) < OPT(|V| - 1, v)$$

Negative weight cycles detection

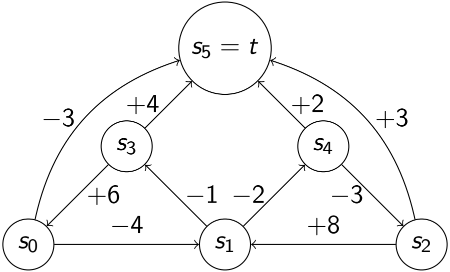
Theorem

G contains a negative cycle **iff**

$$\exists v.\ \ OPT(|V|, v) < OPT(|V| - 1, v)$$

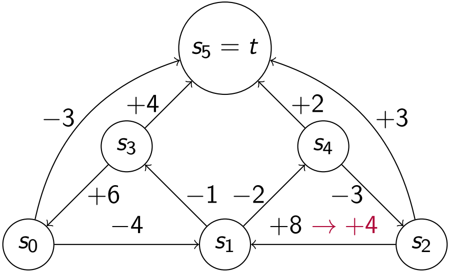
```
# filling the table
for i in range(1,n+1): # filling one more line
    for v in range(n):
        OPT[i][v] = OPT[i-1][v]
        for u in range(n):
            if graph[v][u] != None and \
                    OPT[i][v] > OPT[i-1][u] + graph[v][u]:
                OPT[i][v] = OPT[i-1][u] + graph[v][u]

# Detection of cycles
for v in range(n):
    if OPT[n-1][v] > OPT[n][v]:
        print("Found: negative cycle!\n")
        break;
```

Example



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0
4	-6	-2	3	2	0	0
5	-6	-2	3	0	0	0
6	-6	-2	3	0	0	0

Example



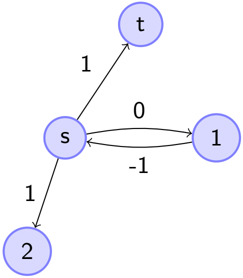
	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0
4	-6	-2	2	2	0	0
5	-6	-2	2	0	-1	0
6	-6	-3	2	0	-1	0

Example

Be careful

As soon as there is a negative weight cycle, the calculated cost for $s \rightarrow t$ may be wrong !

…even if $OPT(|V|-1, s) = OPT(|V|, s)$



	s	1	2	t
0	∞	∞	∞	0
1	1	∞	∞	0
2	1	0	∞	0
3	0	0	∞	0
4	0	-1	∞	0

Demo

Distributed Bellman-Ford

Property

To compute the cost $OPT(i, v)$ for node v at step i, we only need:

- $OPT(i-1, v)$ the value at the previous step for v;
- $OPT(i-1, u)$ the value at the previous step for all the neighbours u of v.

$\rightarrow$ Each node can compute its cost independently, only by communicating with its neighbours

...without being aware of the whole graph!

Distributed Bellman-Ford

Property

To compute the cost $OPT(i, v)$ for node v at step i, we only need:

- $OPT(i-1, v)$ the value at the previous step for v;
- $OPT(i-1, u)$ the value at the previous step for all the neighbours u of v.

$\rightarrow$ Each node can compute its cost independently, only by communicating with its neighbours

…without being aware of the whole graph!

Application in telecommunication networks

Routing problem

Routing in packet-switched communication networks

Problem

Find the best path to route packets up to their destinations.

Routing in packet-switched communication networks

Problem

Find the best path to route packets up to their destinations.

Criteria (weights)

Shortest routes w.r.t number of links, minimal latency, ...

Routing in packet-switched communication networks

Problem

Find the best path to route packets up to their destinations.

Criteria (weights)

Shortest routes w.r.t number of links, minimal latency, ...

Routing specifics

- Each router holds a table (destination, next_router (`Next_Hop`)).
- Computations done locally in routers (without knowing the configuration of the network)

Routing in packet-switched communication networks

Data model (network)

- routers are modeled by graph nodes
- links between routers are modeled by graph arcs
- distances (links numbers, latency) are modeled by arc weights

Routing in packet-switched communication networks

Data model (network)

- routers are modeled by graph nodes
- links between routers are modeled by graph arcs
- distances (links numbers, latency) are modeled by arc weights

Communication

As soon as a router changes its routing table, it warns its neighbours so that they can update their own tables also.

Implementation

Each router runs a loop:

Implementation

Each router runs a loop:

1. wait for a change notification of routing from one of its neighbours

Implementation

Each router runs a loop:

1. wait for a change notification of routing from one of its neighbours
2. recompute its own routing table (Final destination $p \rightarrow Next\_Hop$)

Implementation

Each router runs a loop:

1. wait for a change notification of routing from one of its neighbours
2. recompute its own routing table (Final destination $p \rightarrow Next\_Hop$)
3. send its new distances to its neighbours

Implementation

Each router runs a loop:

1. wait for a change notification of routing from one of its neighbours
2. recompute its own routing table (Final destination $p \rightarrow Next\_Hop$)
3. send its new distances to its neighbours
4. goto 1

Implementation

Each router runs a loop:

1. wait for a change notification of routing from one of its neighbours
2. recompute its own routing table (Final destination $p \rightarrow Next\_Hop$)
3. send its new distances to its neighbours
4. `goto 1`

Each router v keeps locally:

- an array M_v with $M_v[p]$ being the distance of the shortest path between v and p
- an array $\texttt{Next\_Hop}_v$ where $\texttt{Next\_Hop}_v[p]$ is the identifier of the next router for any dispatch towards p

Algorithm for each node v

```
Nv = ... # the list of neighbours of v

def process(u, Mu) :
    """
    At each notification received from a neighbour u
    :param Mu: routing table of u
    """

    # update the local routing table
    update = False
    for p in V:
        if Mu[p] + Timings[v][u] < Mv[p]:
            Mv[p] = Mu[p] + Timings[v][u]
            NextHop_v[p] = u
            update = True

    # notifying neighbours
    if update:
        for u in Nv:
            send_update(u,Mv)
```

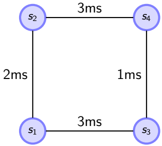
Bellman-Ford algorithm as protocol

Distance Vector Protocol

This protocol is used in computer networks (*e.g.* on Internet)

$\rightarrow$ *Routing Information Protocol* (RIP)

Example



Bellman-Ford algorithm as protocol

Distance Vector Protocol

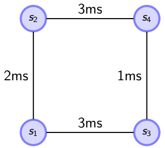
This protocol is used in computer networks (*e.g.* on Internet)

→ *Routing Information Protocol* (RIP)

Example

p	s_1	s_4
$M_2(p)$	2	3
$next(p)$	s_1	s_4

p	s_2	s_3
$M_1(p)$	2	3
$next(p)$	s_2	s_3



p	s_2	s_3
$M_4(p)$	3	1
$next(p)$	s_2	s_3

p	s_1	s_4
$M_3(p)$	3	1
$next(p)$	s_1	s_4

Bellman-Ford algorithm as protocol

Distance Vector Protocol

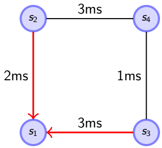
This protocol is used in computer networks (*e.g.* on Internet)

→ *Routing Information Protocol* (RIP)

Example

p	s_1	s_4
$M_2(p)$	2	3
$next(p)$	s_1	s_4

p	s_2	s_3
$M_4(p)$	3	1
$next(p)$	s_2	s_3



p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_4
$M_3(p)$	3	1
$next(p)$	s_1	s_4

Bellman-Ford algorithm as protocol

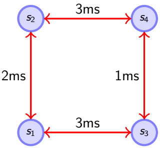
Distance Vector Protocol

This protocol is used in computer networks (*e.g.* on Internet)

$\rightarrow$ *Routing Information Protocol* (RIP)

Example

p	s_1	s_3	s_4
$M_2(p)$	2	4	3
$next(p)$	s_1	s_4	s_4



p	s_1	s_2	s_3
$M_4(p)$	4	3	1
$next(p)$	s_3	s_2	s_3

p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	4	1
$next(p)$	s_1	s_4	s_4

Bellman-Ford algorithm as protocol

Distance Vector Protocol

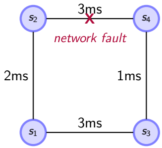
This protocol is used in computer networks (*e.g.* on Internet)

$\rightarrow$ *Routing Information Protocol* (RIP)

Example

p	s_1	s_3	s_4
$M_2(p)$	2	4	3
$next(p)$	s_1	s_4	s_4

p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3



p	s_1	s_2	s_3
$M_4(p)$	4	3	1
$next(p)$	s_3	s_2	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	4	1
$next(p)$	s_1	s_4	s_4

Bellman-Ford algorithm as protocol

Distance Vector Protocol

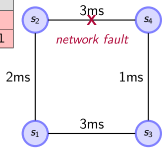
This protocol is used in computer networks (*e.g.* on Internet)

$\rightarrow$ *Routing Information Protocol* (RIP)

Example

p	s_1	s_3	s_4
$M_2(p)$	2	4	∞
$next(p)$	s_1	s_4	null

p	s_1	s_2	s_3
$M_4(p)$	4	∞	1
$next(p)$	s_3	null	s_3



p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	4	1
$next(p)$	s_1	s_4	s_4

Bellman-Ford algorithm as protocol

Distance Vector Protocol

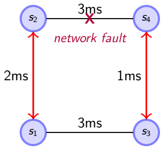
This protocol is used in computer networks (*e.g.* on Internet)

$\rightarrow$ *Routing Information Protocol* (RIP)

Example

p	s_1	s_3	s_4
$M_2(p)$	2	4	6
$next(p)$	s_1	s_4	s_1

p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3



p	s_1	s_2	s_3
$M_4(p)$	4	∞	1
$next(p)$	s_3	null	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	5	1
$next(p)$	s_1	s_1	s_4

Bellman-Ford algorithm as protocol

Distance Vector Protocol

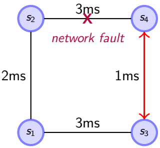
This protocol is used in computer networks (*e.g.* on Internet)

$\rightarrow$ *Routing Information Protocol* (RIP)

Example

p	s_1	s_3	s_4
$M_2(p)$	2	4	6
$next(p)$	s_1	s_4	s_1

p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3



p	s_1	s_2	s_3
$M_4(p)$	4	6	1
$next(p)$	s_3	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	5	1
$next(p)$	s_1	s_1	s_4

Plan

Main points to remember

- « General » resolution method
- Recurrence formula (sub-optimal structure)
- The subproblems are not independent
- Memoization technique: *decrease the execution time by memorizing the calculated values*
  - ➜ Classic compromise in computer science: time *vs* memory
- Generally efficient but not always applicable
- Shortest paths
  - ✗ Be careful with negative weights!
  - ✗ Be careful with negative cycles!
  - ➜ Bellman-Ford algorithm: circumvents these two difficulties
    - Polynomial complexity ($\mathcal{O}(|V|^3)$ or $\mathcal{O}(|V| \times |E|)$)
    - Principle also used for packet routing

Plan

Going further

Concret problem

In bioinformatics (computer science dedicated to biology), sequence alignment allows two biological sequences (DNA, RNA or proteins) to be closer, so as to explain the similar regions.

Example

- Given 2 sequences of any size: the first of size n and the second of size m

$$C \quad T \quad A \quad G \quad C \quad A \quad G \quad T \quad C \quad A$$

$$G \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$$

Example

- Given 2 sequences of any size: the first of size n and the second of size m

$$C \quad T \quad A \quad G \quad C \quad A \quad G \quad T \quad C \quad A$$

$$G \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$$

- an alignment:

$$C \quad T \quad A \quad G \quad C \quad A \quad G \quad - \quad - \quad T \quad C \quad A$$

$$G \quad - \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$$

Example

- Given 2 sequences of any size: the first of size n and the second of size m

$$C \quad T \quad A \quad G \quad C \quad A \quad G \quad T \quad C \quad A$$

$$G \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$$

- an alignment:

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

match

Example

- Given 2 sequences of any size: the first of size n and the second of size m

$$C\ T\ A\ G\ C\ A\ G\ T\ C\ A$$

$$G\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$$

- an alignment:

C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

match   substitution

Example

- Given 2 sequences of any size: the first of size n and the second of size m

$$C \quad T \quad A \quad G \quad C \quad A \quad G \quad T \quad C \quad A$$

$$G \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$$

- an alignment:

C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

match substitution insert/delete

Example

- Given 2 sequences of any size: the first of size n and the second of size m

$$C \quad T \quad A \quad G \quad C \quad A \quad G \quad T \quad C \quad A$$

$$G \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$$

- an alignment:

C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

match substitution insert/delete

- an other alignment:

C	T	A	G	C	A	G	T	C	A	–	–	–
–	G	A	G	C	A	–	T	C	A	T	C	G

Which alignment to chose?

- to each elementary operation we associate a score:
  - match : 1
  - substitution : -1
  - insert/delete : -2

Which alignment to chose?

- to each elementary operation we associate a score:
  - match : 1
  - substitution : -1
  - insert/delete : -2

The score of an alignment is the sum of the elementary scores

Which alignment to chose?

- to each elementary operation we associate a score:
  - match : 1
  - substitution : -1
  - insert/delete : -2

The score of an alignment is the sum of the elementary scores

- first alignment: -3

C	T	A	G	C	A	G	$-$	$-$	T	C	A
G	$-$	A	G	C	A	T	C	A	T	C	G
-1	-2	1	1	1	1	-1	-2	-2	1	1	-1

Which alignment to chose?

- to each elementary operation we associate a score:
  - match : 1
  - substitution : -1
  - insert/delete : -2

The score of an alignment is the sum of the elementary scores

- first alignment: -3

C	T	A	G	C	A	G	—	—	T	C	A
G	—	A	G	C	A	T	C	A	T	C	G
-1	-2	1	1	1	1	-1	-2	-2	1	1	-1

- second alignment: -4

C	T	A	G	C	A	G	T	C	A	—	—	—
—	G	A	G	C	A	—	T	C	A	T	C	G

Optimization problem

Sequence alignment

Given :

- 2 sequences
- 3 scores associated to the 3 elementary operations (match, subst, ins/del)

Optimization problem

Sequence alignment

Given :

- 2 sequences
- 3 scores associated to the 3 elementary operations (match, subst, ins/del)

Problem : find the alignment with the maximal score

Optimization problem

Sequence alignment

Given :

- 2 sequences
- 3 scores associated to the 3 elementary operations (match, subst, ins/del)

Problem : find the alignment with the maximal score

Exponential Complexity!

Number of alignments: $\sum_{i=0}^{n} C_{m+i}^{i} \times C_{m}^{n-i} = \mathcal{O}(2^{n+m})$

Recursive Approach

$$
\begin{aligned}
Align([],[]) &= 0\\
Align(S[0{:}n],[]) &= n \times score['ins/del']\\
Align([],T[0{:}m]) &= m \times score['ins/del']\\
Align(S[0{:}n],T[0{:}m]) &= max \left\{ \vphantom{\begin{matrix}a\\b\\c\\d\end{matrix}} \right.
\end{aligned}
$$

Recursive Approach

$$
\begin{aligned}
Align([],[]) &= 0 \\
Align(S[0{:}n],[]) &= n \times score['ins/del'] \\
Align([],T[0{:}m]) &= m \times score['ins/del'] \\
Align(S[0{:}n],T[0{:}m]) &= max \left\{ \right.
\end{aligned}
$$

S :	C	T	A	G	C	A	G	T	C	A	
T :	G	A	G	C	A	T	C	A	T	C	G

Recursive Approach

$$
\begin{aligned}
Align([],[]) &= 0 \\
Align(S[0{:}n],[]) &= n \times score['ins/del'] \\
Align([],T[0{:}m]) &= m \times score['ins/del'] \\
Align(S[0{:}n],T[0{:}m]) &= max \begin{cases} Align(S[0{:}n],T[0{:}m-1]) + score['ins/del'] \\ \\ \\ \end{cases}
\end{aligned}
$$

S :	C	T	A	G	C	A	G	T	C	A	
T :	G	A	G	C	A	T	C	A	T	C	G

C	T	A	G	C	A	G	T	C	A	$-$
G	A	G	C	A	T	C	A	T	C	G
										-2

Recursive Approach

$$
\begin{aligned}
&Align([],[]) &&= 0\\
&Align(S[0{:}n],[]) &&= n \times score['ins/del']\\
&Align([],T[0{:}m]) &&= m \times score['ins/del']\\
&Align(S[0{:}n],T[0{:}m]) &&= max \begin{cases} Align(S[0{:}n],T[0{:}m-1]) + score['ins/del'] \\ Align(S[0{:}n-1],T[0{:}m]) + score['ins/del'] \\ \\ \end{cases}
\end{aligned}
$$

S : C T A G C A G T C A
T : G A G C A T C A T C G

C	T	A	G	C	A	G	T	C	A	–
G	A	G	C	A	T	C	A	T	C	G
										-2

C	T	A	G	C	A	G	T	C		A
G	A	G	C	A	T	C	A	T	C	G	–
											-2

Recursive Approach

$$
\begin{aligned}
&Align([],[]) &&= 0\\
&Align(S[0{:}n],[]) &&= n \times score['ins/del']\\
&Align([],T[0{:}m]) &&= m \times score['ins/del']\\
&Align(S[0{:}n],T[0{:}m]) &&= max \begin{cases} Align(S[0{:}n],T[0{:}m-1]) + score['ins/del'] \\ Align(S[0{:}n-1],T[0{:}m]) + score['ins/del'] \\ Align(S[0{:}n-1],T[0{:}m-1]) + \begin{cases} score['match'] \; si \;\; S[n] = T[m] \\ score['subst'] \;\; si \;\; S[n] \neq T[m] \end{cases} \end{cases}
\end{aligned}
$$

S :	C	T	A	G	C	A	G	T	C	A	
T :	G	A	G	C	A	T	C	A	T	C	G

C	T	A	G	C	A	G	T	C	A		–
G	A	G	C	A	T	C	A	T	C		G
											-2

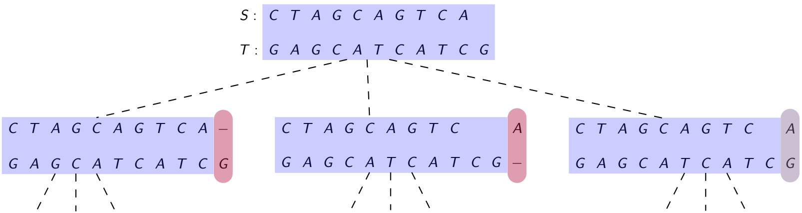
C	T	A	G	C	A	G	T	C			A
G	A	G	C	A	T	C	A	T	C	G	–
											-2

C	T	A	G	C	A	G	T	C		A
G	A	G	C	A	T	C	A	T	C	G
										-1

Recursive approach

Exponential complexity!

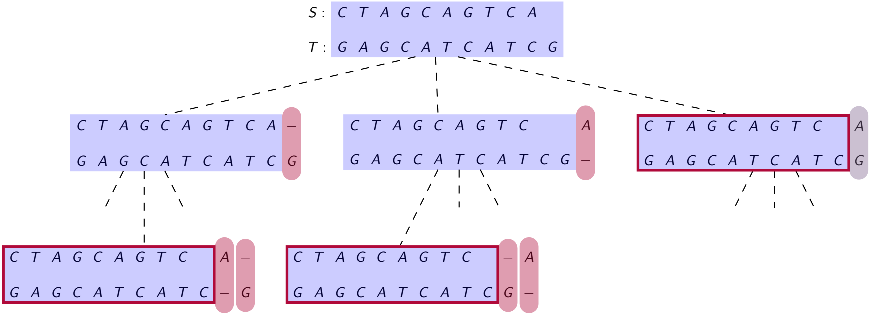
- Ternary search tree of depth $n + m$
- Complexity in $\mathcal{O}(3^{n+m})$



Recursive approach

Exponential complexity!

- Ternary search tree of depth $n + m$
- Complexity in $\mathcal{O}(3^{n+m})$
- Redundant computation!



Recurrence formula

Sequence alignment

Let $OPT(M, N)$ be the maximal score
of the alignment of the **M** first nucleotides of the first sequence
with the **N** first nucleotides of the second sequence

- $OPT(0, 0) = 0$
- $OPT(N, M) =$ maximum among:
  $OPT(N-1, M) + (-2)$ if insert N^{e}
  $OPT(N, M-1) + (-2)$ if insert M^{e}
  $OPT(N-1, M-1) + (\pm 1)$ depending on match or supp

Needleman and Wunsch algorithm (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$C\ T\ A\ G\ C\ A\ G\ -\ -\ T\ C\ A$

$G\ -\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$

Needleman and Wunsch algorithm (1970)

skip

$S \backslash T$		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$C \quad T \quad A \quad G \quad C \quad A \quad G \quad - \quad - \quad T \quad C \quad A$

$G \quad - \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$

Needleman and Wunsch algorithm (1970)

skip

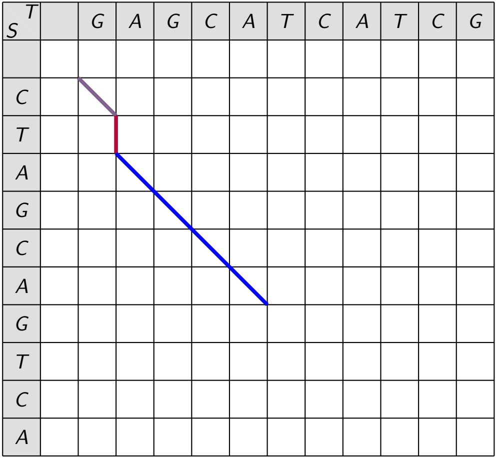
S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$C \quad T \quad A \quad G \quad C \quad A \quad G \quad - \quad - \quad T \quad C \quad A$

$G \quad - \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$

Needleman and Wunsch algorithm (1970)

skip

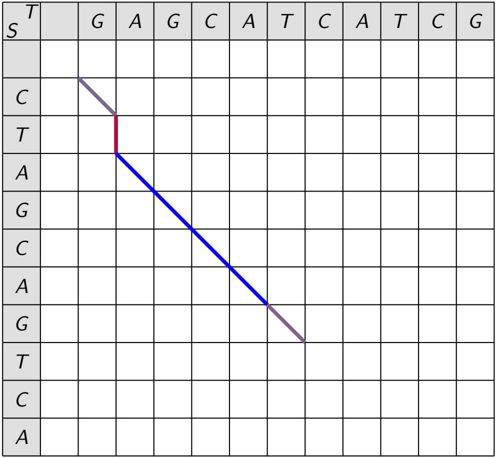


$C \quad T \quad A \quad G \quad C \quad A \quad G \quad - \quad - \quad T \quad C \quad A$

$G \quad - \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$

Needleman and Wunsch algorithm (1970)

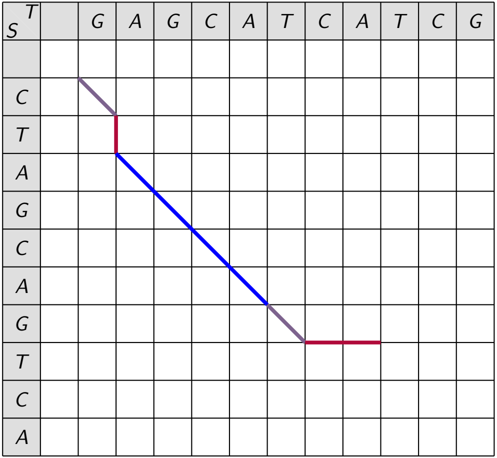
▸▸ skip



$C \quad T \quad A \quad G \quad C \quad A \quad G \quad - \quad - \quad T \quad C \quad A$

$G \quad - \quad A \quad G \quad C \quad A \quad T \quad C \quad A \quad T \quad C \quad G$

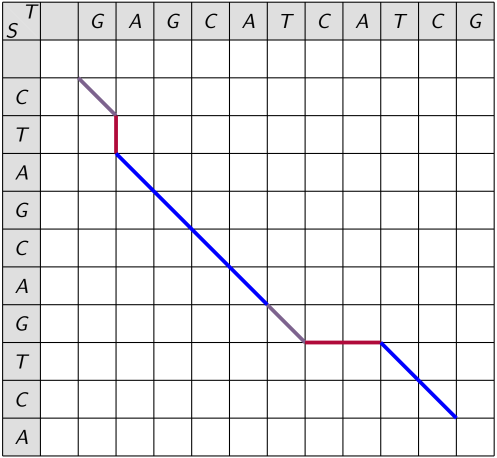
Needleman and Wunsch algorithm (1970)



C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

Needleman and Wunsch algorithm (1970)

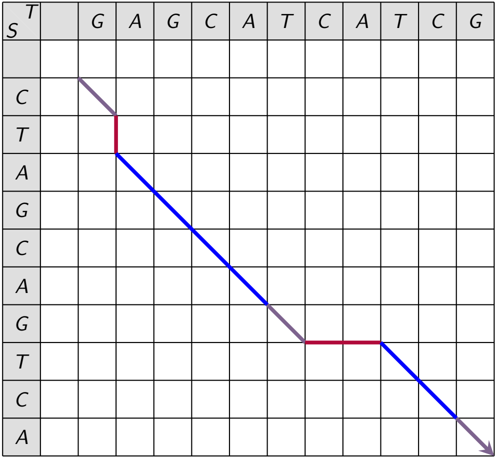
skip



C	*T*	*A*	*G*	*C*	*A*	*G*	–	–	*T*	*C*	*A*
G	–	*A*	*G*	*C*	*A*	*T*	*C*	*A*	*T*	*C*	*G*

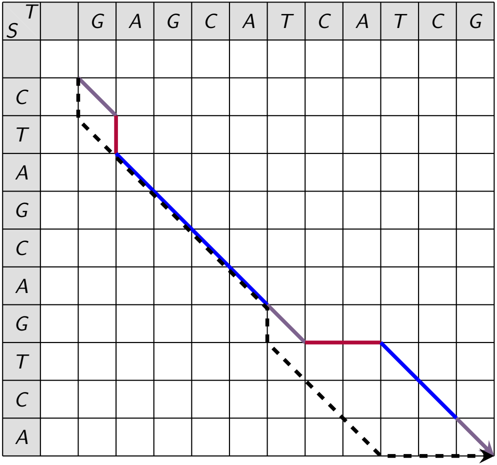
Needleman and Wunsch algorithm (1970)

skip



C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

Needleman and Wunsch algorithm (1970)



C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

C T A G C A G T C A – – –

– G A G C A – T C A T C G

Needleman and Wunsch algorithm (1970)

skip

T / S		G	A	G	C	A	T	C	A	T	C	G
	0											
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

Goal: to find the alignment with the maximal score

Needleman and Wunsch algorithm (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	→ −2	→ −4	→ −6	→ −8	→ −10	→ −12	→ −14	→ −16	→ −18	→ −20	→ −22
C	↓ −2											
T	↓ −4											
A	↓ −6											
G	↓ −8											
C	↓ −10											
A	↓ −12											
G	↓ −14											
T	↓ −16											
C	↓ −18											
A	↓ −20											

Step 1: we fill the first line and the first column

- here score['ins/del'] = -2 (→)

Needleman and Wunsch algorithm (1970)

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

skip

S \ T		G
	0	−2
C	−2	

Step 2: we fill every cells by maximizing on the 3 axes

- here score['match'] = 1 (→) and score['subst'] = -1 (→)

Needleman and Wunsch algorithm (1970)

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

skip

S \ T		G
	0	−2
C	−2	−4

Step 2: we fill every cells by maximizing on the 3 axes

- here score['match'] = 1 (→) and score['subst'] = -1 (→)

Needleman and Wunsch algorithm (1970)

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

skip

S \ T		G
	0	−2
C	−2	−1

Step 2: we fill every cells by maximizing on the 3 axes

- here score['match'] = 1 (→) and score['subst'] = -1 (→)

Needleman and Wunsch algorithm (1970)

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

S \ T		G
	0	−2
C	−2	−4

Step 2: we fill every cells by maximizing on the 3 axes

- here score['match'] = 1 ($\rightarrow$) and score['subst'] = -1 ($\rightarrow$)

Needleman and Wunsch algorithm (1970)

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	−1										
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

S \ T		G
	0	−2
C	−2	−1

Step 2: we fill every cells by maximizing on the 3 axes

- here score['match'] = 1 (→) and score['subst'] = -1 (→)

Needleman and Wunsch algorithm (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	-1	-3	-5	-5	-7	-9	-11	-13	-15	-17	-19
T	−4	-3	-2	-4	-6	-6	-6	-8	-10	-12	-14	-16
A	−6	-5	-2	-3	-5	-5	-7	-7	-7	-9	-11	-13
G	−8	-5	-4	-1	-3	-6	-6	-8	-8	-8	-10	-10
C	−10	-7	-6	-3	0	-2	-4	-5	-7	-9	-7	-9
A	−12	-9	-6	-5	-2	1	-1	-3	-4	-6	-8	-8
G	−14	-11	-8	-5	-4	-1	0	-2	-4	-5	-7	-7
T	−16	-13	-10	-7	-6	-3	0	-1	-3	-3	-5	-7
C	−18	-15	-12	-9	-6	-5	-2	1	-1	-3	-2	-4
A	−20	-17	-14	-11	-8	-5	-4	-1	2	0	-2	

Step 2: we fill every cells by maximizing on the 3 axes

- here score['match'] = 1 (→) and score['subst'] = -1 (→)

Needleman and Wunsch algorithm (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	-1	-3	-5	-5	-7	-9	-11	-13	-15	-17	-19
T	−4	-3	-2	-4	-6	-6	-6	-8	-10	-12	-14	-16
A	−6	-5	-2	-3	-5	-5	-7	-7	-7	-9	-11	-13
G	−8	-5	-4	-1	-3	-6	-6	-8	-8	-8	-10	-10
C	−10	-7	-6	-3	0	-2	-4	-5	-7	-9	-7	-9
A	−12	-9	-6	-5	-2	1	-1	-3	-4	-6	-8	-8
G	−14	-11	-8	-5	-4	-1	0	-2	-4	-5	-7	-7
T	−16	-13	-10	-7	-6	-3	0	-1	-3	-3	-5	-7
C	−18	-15	-12	-9	-6	-5	-2	1	-1	-3	-2	-4
A	−20	-17	-14	-11	-8	-5	-4	-1	2	0	-2	-3

Step 3: at the end of the table, we get the maximal score and the optimal alignments

Needleman and Wunsch algorithm (1970)

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	-1	-3	-5	-5	-7	-9	-11	-13	-15	-17	-19
T	−4	-3	-2	-4	-6	-6	-6	-8	-10	-12	-14	-16
A	−6	-5	-2	-3	-5	-5	-7	-7	-7	-9	-11	-13
G	−8	-5	-4	-1	-3	-6	-6	-8	-8	-8	-10	-10
C	−10	-7	-6	-3	0	-2	-4	-5	-7	-9	-7	-9
A	−12	-9	-6	-5	-2	1	-1	-3	-4	-6	-8	-8
G	−14	-11	-8	-5	-4	-1	0	-2	-4	-5	-7	-7
T	−16	-13	-10	-7	-6	-3	0	-1	-3	-3	-5	-7
C	−18	-15	-12	-9	-6	-5	-2	1	-1	-3	-2	-4
A	−20	-17	-14	-11	-8	-5	-4	-1	2	0	-2	-3

Step 3: at the end of the table, we get the maximal score and the optimal alignments

Needleman and Wunsch algorithm (1970)

- Among the 3^{n+m} possible paths in the matrix, we found the optimal alignment in $n \times m$ steps

Needleman and Wunsch algorithm (1970)

- Among the 3^{n+m} possible paths in the matrix, we found the optimal alignment in $n \times m$ steps

Algorithm complexity

- $\mathcal{O}(n \times m)$ in time: size of the matrix
- $\mathcal{O}(min(n, m))$ in space: instead of keeping the complete matrix, we keep only the current and precedent line