Information, Computation, and Communication

Algorithm 2

What is the complexity of this algorithm?

Algorithm2

input : *a list L with n numbers*
output : *a number s*

```
s ⟵ 0
i ⟵ 1
while i < n
    for j from 1 to n
        s ⟵ s + L[j]
    i ⟵ 2 · i
return : s
```

- How many iterations do the **for** and the **while loops** have?
- The **for loop** runs from 1 to n → n iterations/repetitions
- The **while loop** repeats if i < n. What is the value of i after iteration k?
- i is multiplied by 2 in every iteration → the value of i after iteration k is 2^k
- So, after which iteration is $2^k \geq n$?
- If $k \geq \log_2(n)$
- So, the while loop has $\log_2(n)$ iterations

$$\Theta(n \cdot \log(n))$$

Topics

- Sorting (selection sort)
- Recursion
  - Complexity of a recursive algorithm
  - Binary Search (« Recherche dichotomique »)
  - Merge Sort
  - Fibonacci numbers
  - Dynamic Programming
- (Application of Binary Search)

Sorting

- There are many different sorting algorithms (selection sort, insertion sort, bubble sort, merge sort ..)
- Input: a list L of elements, e.g., integers
- Output a sorted version of L

Selection Sort

- Divide the list into two parts:
  - A sorted part (on the left, initially empty)
  - An unsorted part (on the right, initially the whole list)
- Find the smallest element in the unsorted part and put it at the end of the sorted part by swapping two elements, e.g., {5,4,6,1,2,7,8,3}
- After 1st iteration: {1,4,6,5,2,7,8,3}
- After 2nd iteration: {1,2,6,5,4,7,8,3}
- After 3rd iteration: {1,2,3,5,4,7,8,6}

Selection Sort

- Divide the problem:
  - Find the smallest element in a list
  - Put the element in the right place by swapping two elements

Recall Maximal Value from Last Week

Maximal Value

input : *(non-empty) liste L with n integers*
output : *the largest value in the list*

```
x_max ⟵ L[1]
for i from 2 to n
    if L[i] > x_max
        x_max ⟵ L[i]
return : x_max
```

Let's create a variant to find the position of the minimal value:

find_minimum_position (L)

input : *a non-empty list L of numbers*
output : *the position of the smallest number in L*

```
n ⟵ size(L)
pos ⟵ 1
x ⟵ L[1]
for i from 2 to n
    if L[i] < x
        pos ⟵ i
        x ⟵ L[i]
return : pos
```

What is the complexity of this variant?

We will use **size(L)** to refer to the length of the list L.
We assume **size(L) is in Θ(1).**

Swap Two Elements in a List

swap(L, i, j)

input : *a list L of numbers and two indices i, j*
output : *the list L in which the values at index i and j are swapped*

$n \longleftarrow$ **size**(L)
if $i \leq n$ *and* $j \leq n$ *(and* $i \geq 1$ *and* $j \geq 1$*)*
    $x \longleftarrow L[i]$
    $L[i] \longleftarrow L[j]$
    $L[j] \longleftarrow x$
return : L

What is the complexity of this algorithm?

Selection Sort

sort(L)

input : *a list L of numbers*
output : *a sorted version of L*

```
n ⟵ size(L)
for i from 1 to n
    index ⟵ find_minimum_position(L(i : n))
    swap(L, i, index)
return : L
```

Line	Costs	Repet.
1	~ 1	1
2	~ 1	n
3	~ n	n
4	~ 1	n
5	~ 1	1

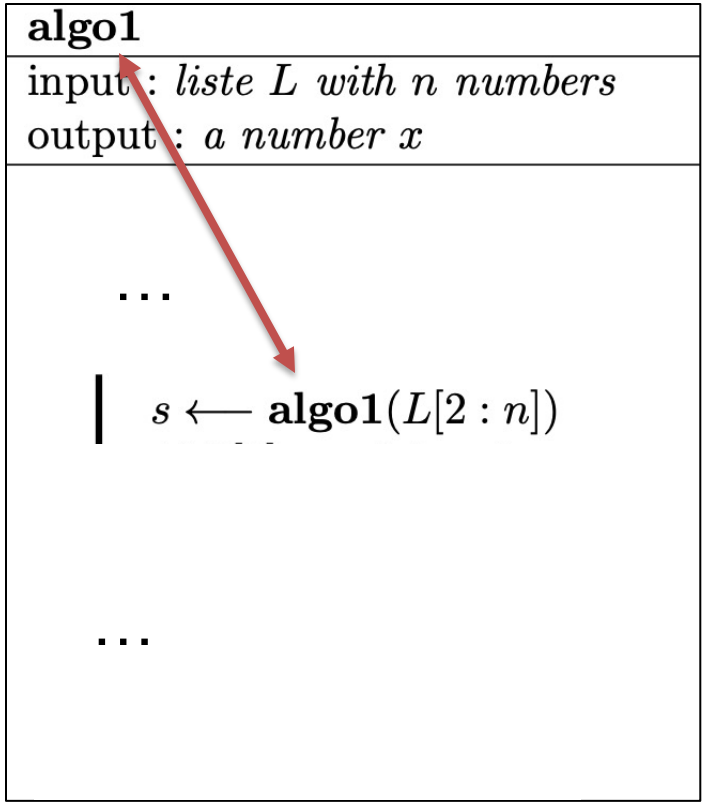
Sum over the lines: $\sum_{i=1}^{5} c(i) \cdot r(i) = 1 + 2n + n^2 + 1 = \Theta(n^2)$

Sum over iterations of line 3: $\sum_{i=1}^{n} (n + 1 - i) = \sum_{i=1}^{n} n + \sum_{i=1}^{n} 1 - \sum_{i=1}^{n} i = n^2 + n - \frac{n \cdot (n+1)}{2} = \Theta(n^2)$

Size of the list in iteration i

Recursive Algorithm

- An algorithm that calls itself



Your Tasks

- Understand a recursive algorithm
- Analyze the complexity of a recursive algorithm
- Write recursive algorithm

Understanding a Recurive Algorithm

algo1

input : *liste L with n numbers*
output : *a number x*

```
if n = 0
    x ⟵ 0
else
    s ⟵ algo1(L[2 : n])
    if L[1] mod 2 = 0
        x ⟵ s + L[1]
    else
        x ⟵ s
return : x
```

What is the output if L={5,8,6,10,3}?

Understanding a Recurive Algorithm

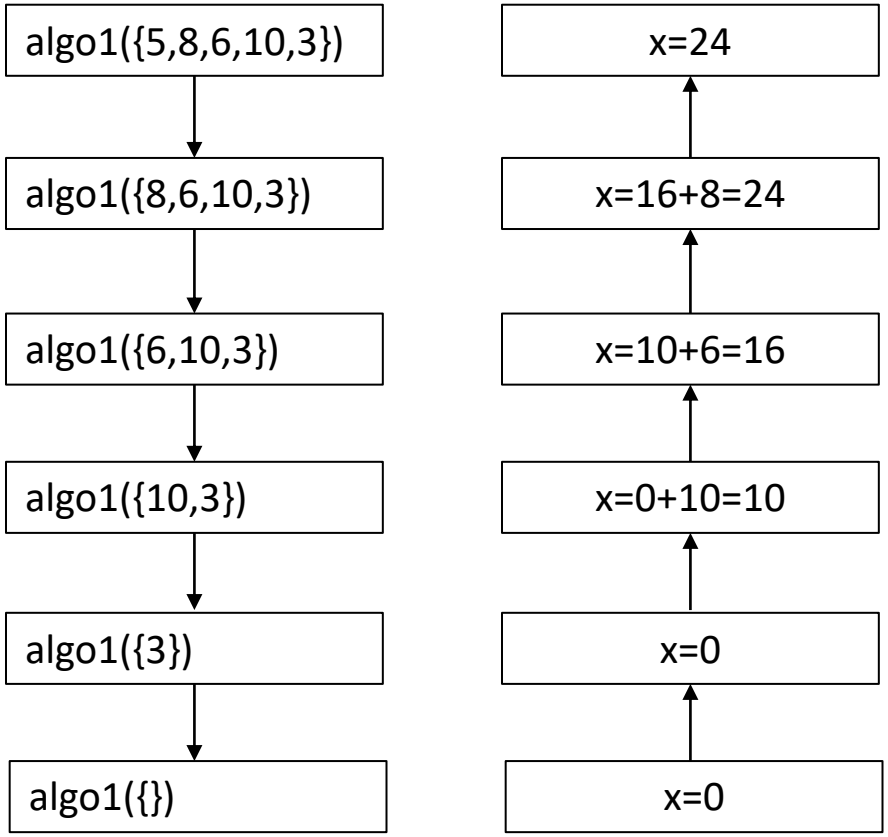
algo1

input : *liste L with n numbers*
output : *a number x*

```
if n = 0
    x ⟵ 0
else
    s ⟵ algo1(L[2 : n])
    if L[1] mod 2 = 0
        x ⟵ s + L[1]
    else
        x ⟵ s
return : x
```

What is the output if L={5,8,6,10,3}?

Function **call tree**:

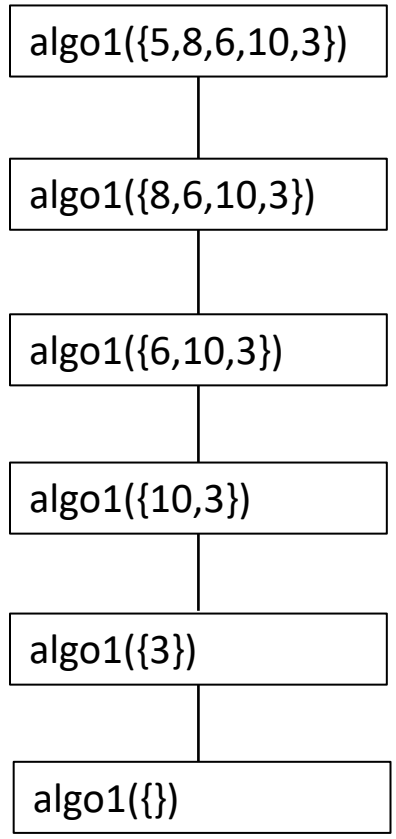


Complexity

algo1

input : *liste L with n numbers*
output : *a number x*

```
if n = 0
    x ⟵ 0
else
    s ⟵ algo1(L[2 : n])
    if L[1] mod 2 = 0
        x ⟵ s + L[1]
    else
        x ⟵ s
return : x
```



Height? Cost?

algo1({5,8,6,10,3})	~ 10
algo1({8,6,10,3})	--"--
algo1({6,10,3})	--"--
algo1({10,3})	--"--
algo1({3})	--"--
algo1({})	3

n+1

T(n)=10 · n + 3 = **Θ(n)**

T(n)= T(n-1) + 10
T(0) = 3

n+1 nodes · const. cost per node = n+1 = **Θ(n)**

Writing a Recursive Algorithm

- Find the largest number in a list
  - Input: a list L with n numbers
  - Output: the largest number in this list
- Idea of recursion:
  - Use a solution of a smaller problem (a shorter list)
- You need to answer two questions:
  1. Assume we are given the largest element in the list L[2:n] (list of length n-1), how would we compute the largest element of the list L[1:n]?
  2. What is the termination condition? What is the largest element of a list of size 1?

Question 1: from n-1 to n

rec_max

input : *(non-empty) liste L with n numbers*
output : *the maximal number in the list*

$max \longleftarrow \mathbf{rec\_max}(L[2:n])$
if $L[1] > max$
  $max \longleftarrow L[1]$

return : max

Question 2: Termination

rec_max

input : *(non-empty) liste L with n numbers*
output : *the maximal number in the list*

if $n = 1$
    $max \longleftarrow L[1]$
else
return : max

Find Max (Recursively)

rec_max

input : *(non-empty) liste L with n numbers*
output : *the maximal number in the list*

```
if n = 1
    max ⟵ L[1]
else
    max ⟵ rec_max(L[2 : n])
    if L[1] > max
        max ⟵ L[1]
return : max
```

What is the complexity of this algorithm?

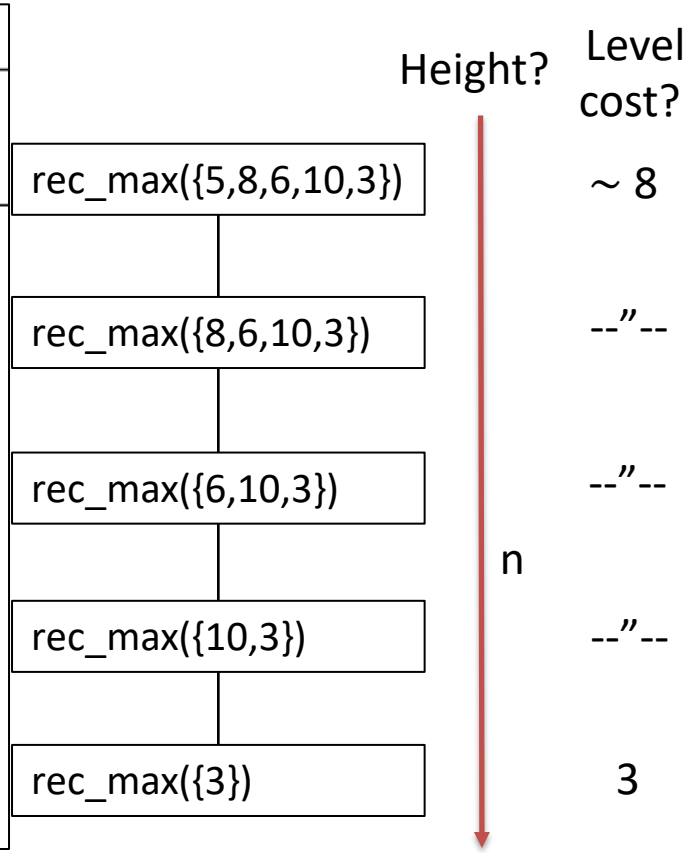
A. Θ(log(n))
B. Θ(n)
C. $\Theta(n^2)$
D. $\Theta(2^n)$

Complexity

rec_max

input : *(non-empty) liste L with n numbers*
output : *the maximal number in the list*

```
if n = 1
    max ⟵ L[1]
else
    max ⟵ rec_max(L[2 : n])
    if L[1] > max
        max ⟵ L[1]
return : max
```



Sum of costs over all levels: T(n)= 8 · (n-1) + 3 = **Θ(n)**

(Recall see in week 6 in programming)

Binary Search – Recherche dichotomique

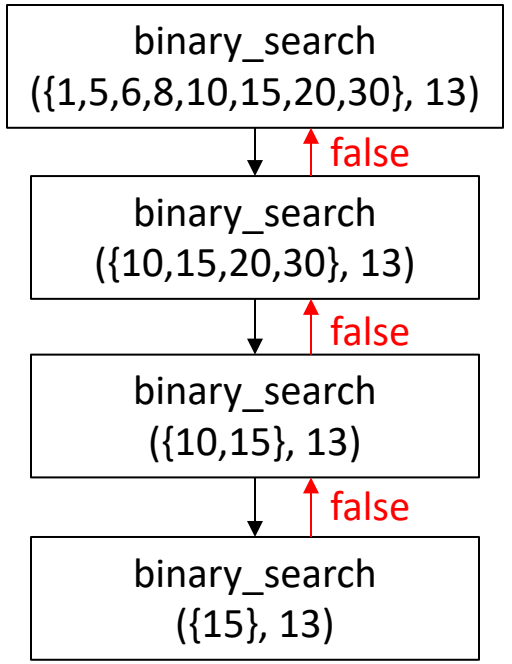
- Find an element in a sorted list
  - Input: a sorted list L of numbers and a number x
  - Output: true, if x is in L, otherwise false
- Idea:
  - Look at the number y in the middle of L
  - If x <= y, repeat the search in the left part of the list
  - If x > y, repeat the search in the right part of the list
  - Termination: x found or list empty.

Binary Search

binary_search(L,x)

input : *a sorted list L of numbers and a number x*
output : *true, if x ∈ L, otherwise false*

```
n ⟵ size(L)
if n = 0
    f ⟵ false
if n = 1
    f ⟵ (L[1] = x)
if n > 1
    m ⟵ ⌊n/2⌋
    if x ≤ L[m]
        f ⟵ binary_search(L[1 : m])
    else
        f ⟵ binary_search(L[m + 1 : n])
return : f
```



Θ(log n)

Height?

Level cost?

~ 10

--"--

--"--

~ 6

Application of Binary Search: Inspection of Exam Results

- About 300 copies sorted by SCIPER number
- How many copies do you have to look at (in the worst case) in order to find your copy?

Application of Binary Search: Inspection of Exam Results

- Roughly 9 copies:
  1. Split the pile into two piles of about half the size (~150 copies)
  2. Check the SCIPER number of the copy on the top of the second pile:
     a) if it is your copy, you are done
     b) if it is not your copy and if your SCIPER number is larger than the one of the copy continue your search in the second pile by going to Step 1,
     c) otherwise continue the search in the first pile by going to Step 1
- Approx. sizes of search piles:
  300, 150, 75, 38, 19, 10, 5, 3, 2, 1

Merge Sort

- Idea:
  - Split the list in half
  - Sort each half
  - Merge the two sorted halfs
- Decompose the problem
  - Split list
  - Merge two sorted list

Recursive Merge

merge

input : *two sorted lists* L *and* R
output : *a sorted list with all the elements in* L *and* R

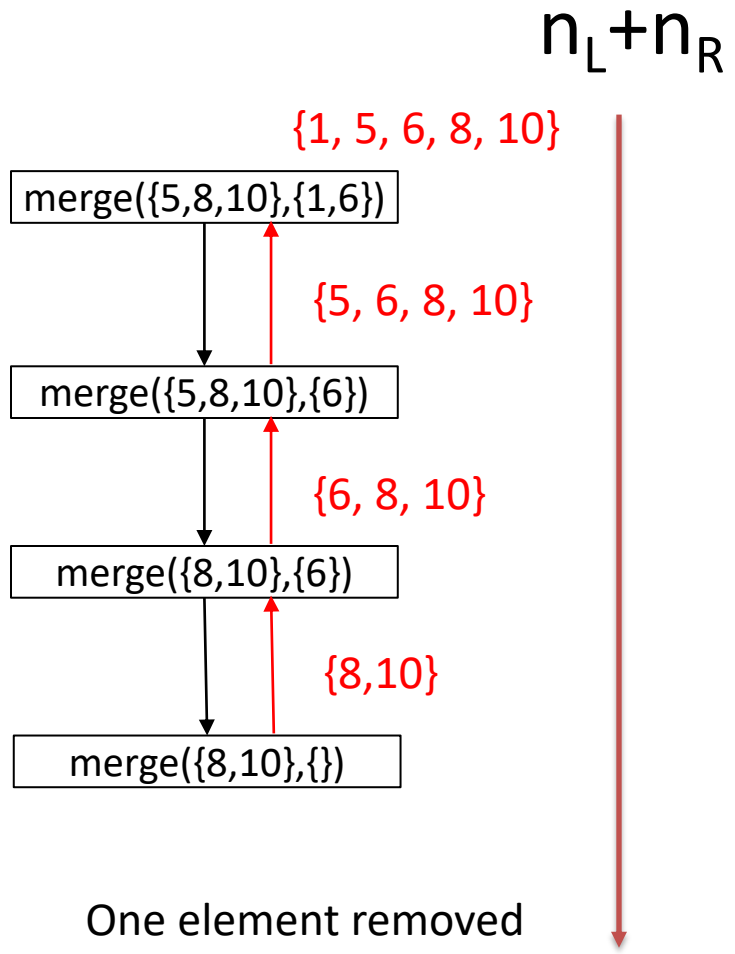
$n_L \longleftarrow \textbf{size}(L)$
$n_R \longleftarrow \textbf{size}(R)$
if $n_L = 0$
    return R
if $n_R = 0$
    return L
if $L[1] < R[1]$
    $M \longleftarrow \textbf{merge}(L[2 : n_L], R)$
    $M \longleftarrow L[1] : M$ *//prepend L[1] to M*
else
    $M \longleftarrow \textbf{merge}(L, R[2 : n_L])$
    $M \longleftarrow R[1] : M)$
return M

Merge

merge

input : *two sorted lists L and R*
output : *a sorted list with all the elements in L and R*

$n_L \longleftarrow \textbf{size}(L)$
$n_R \longleftarrow \textbf{size}(R)$
if $n_L = 0$
  return R
if $n_R = 0$
  return L
if $L[1] < R[1]$
  $M \longleftarrow \textbf{merge}(L[2 : n_L], R)$
  $M \longleftarrow L[1] : M$ *//prepend L[1] to M*
else
  $M \longleftarrow \textbf{merge}(L, R[2 : n_L])$
  $M \longleftarrow R[1] : M)$
return M



One element removed at every call

$\Theta(n_L + n_R)$

Exercise: Write a non-recursive version

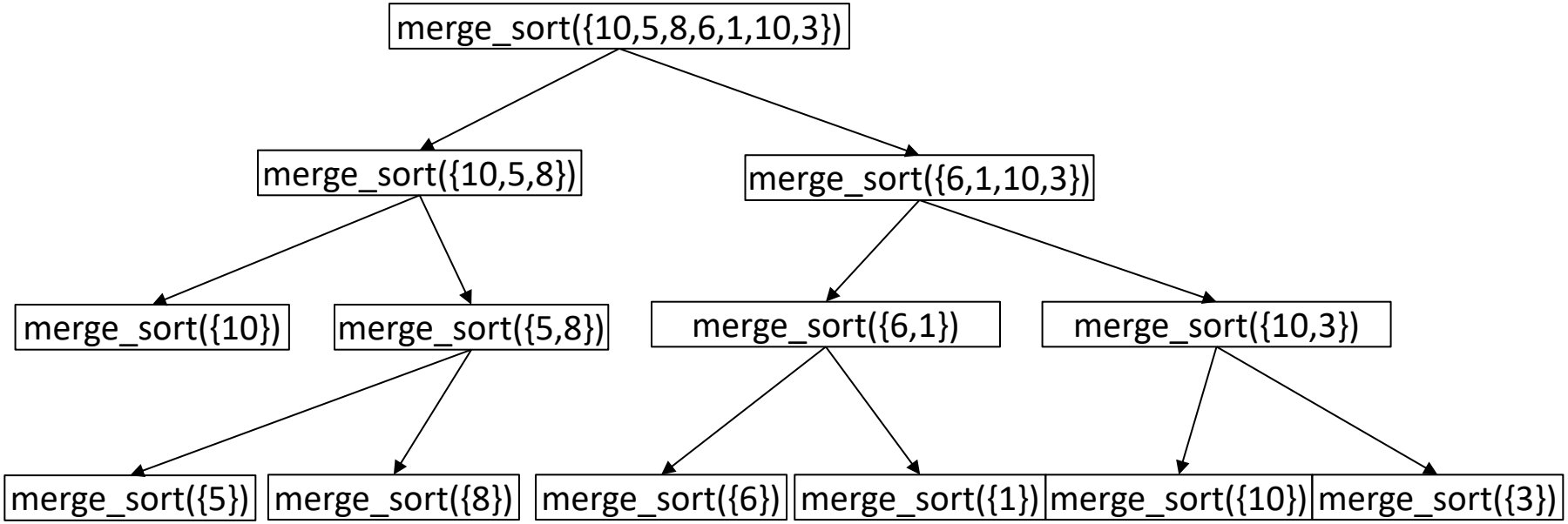
Merge Sort

merge_sort(L)

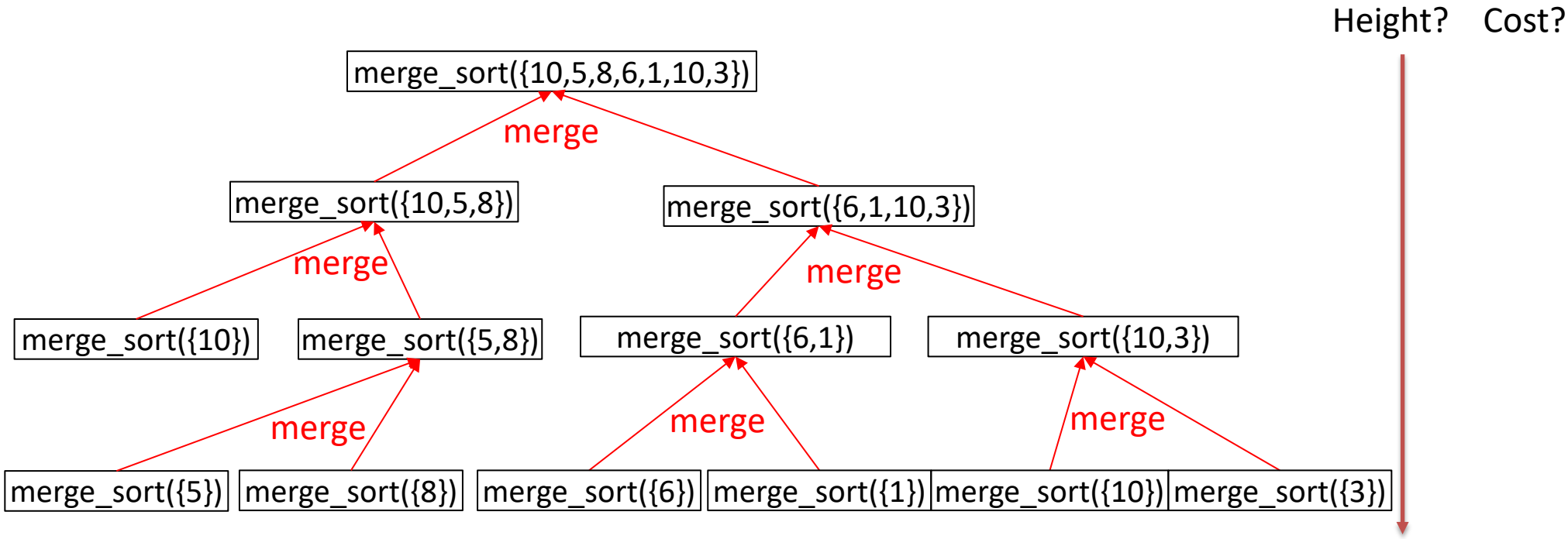
input : *a list L of numbers*
output : *a sorted version of L*

```
n ⟵ size(L)
if n > 1
    m ⟵ ⌊n/2⌋
    left ⟵ merge_sort(L[1 : m])
    right ⟵ merge_sort(L[m + 1 : n])
    L ⟵ merge(left, right)
return : L
```

Merge Sort


merge_sort({10,5,8,6,1,10,3})
merge_sort({10,5,8})
merge_sort({6,1,10,3})
merge_sort({10})
merge_sort({5,8})
merge_sort({6,1})
merge_sort({10,3})
merge_sort({5})
merge_sort({8})
merge_sort({6})
merge_sort({1})
merge_sort({10})
merge_sort({3})

Merge Sort



- Height: how often can we divide n by 2? The height is log n.
- How many nodes? $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 = 2 \cdot 2^{\log n} - 1 = 2n - 1$
- Costs vary per node but we can bound the costs per level.
  At each level, we merge at most n elements.

$\Theta(n \cdot \log n)$

Recursion or Not?

- The recursive solution is not always the only solution and **rarely the most efficient**...
- ...but it is sometimes much simpler and more practical to implement !
- Examples : sorting, processing of recursive data structures (e.g. trees, graphs, ...), ...

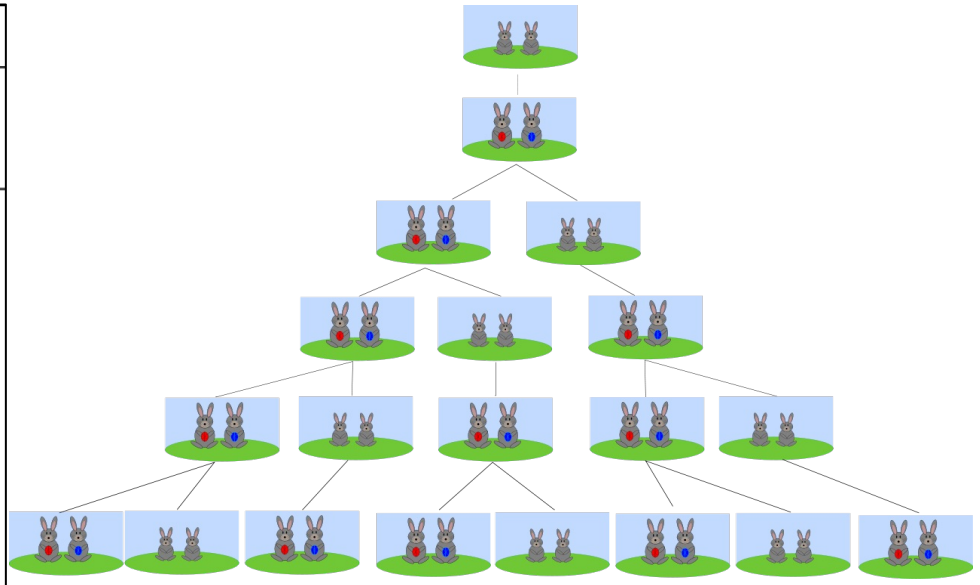
Fibonacci Numbers (Recursive Version)

$F(n) = F(n-1) + F(n-2)$ for $n > 1$
$F(0) = 0$ and $F(1) = 1$

Fibonacci

input : *a integer n*
output : *the Fibonacci number of n*

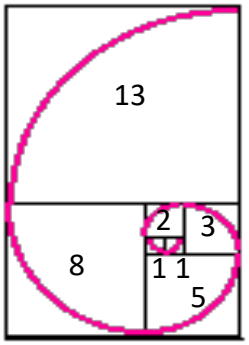
```
if n = 0
    f ⟵ 0
if n = 1
    f ⟵ 1
if n > 1
    p ⟵ Fibonacci(n − 1)
    q ⟵ Fibonacci(n − 2)
    f ⟵ p + q
return : f
```



Month 1: 1
Month 2: 1
Month 3: 2
Month 4: 3
Month 5: 5
Month 6: 8

Named after Leonardo Fibonacci (1175-1250)

Fibonacci Numbers in Nature



The lengths of the squares describing naturally appear spirals are often Fibonacci numbers.

Number of ancestors of a drone (a male honey bee) is a sum of Fibonacci number.

In these pictures there are 55 curves of seeds spiraling to the left as you go outwards and 34 spirals of seeds spiraling to the right. A little further towards the center you can count 34 spirals to the left and 21 spirals to the right. These pairs of numbers are (almost always) neighbors in the Fibonacci series.

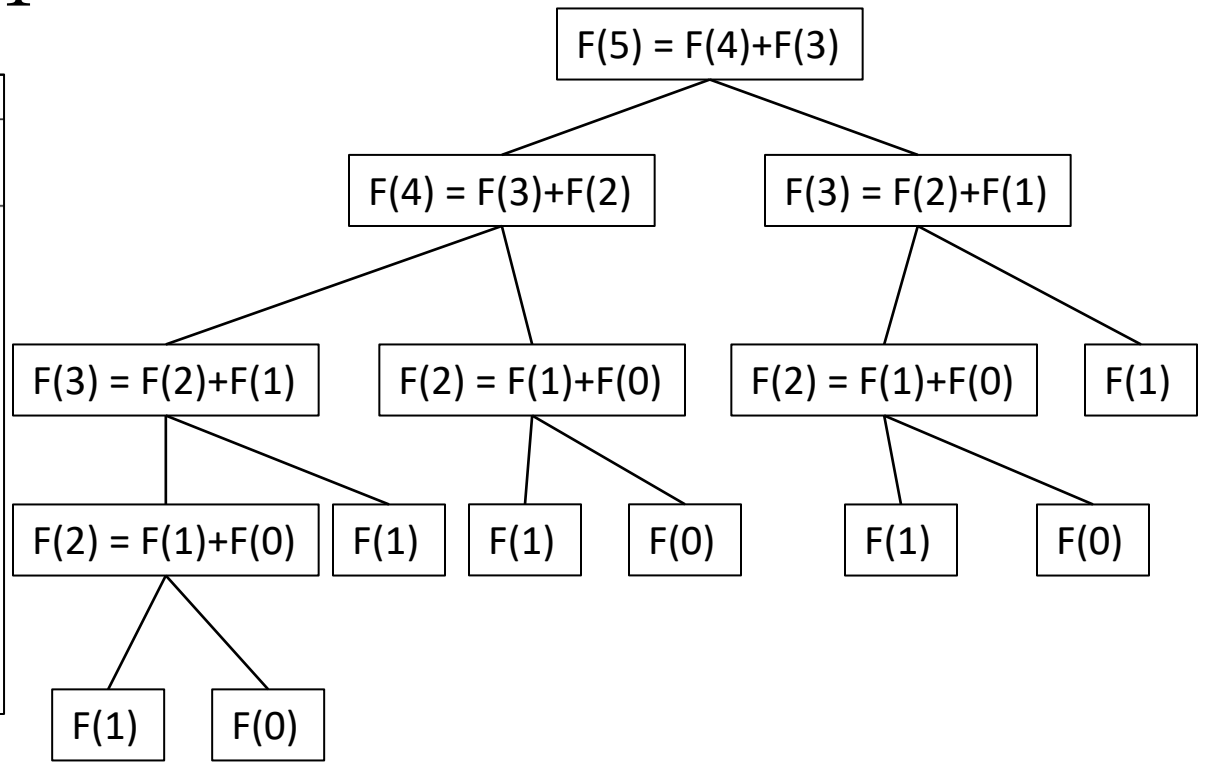
https://plus.maths.org/content/life-and-numbers-fibonacci

Fibonacci Recursive

$F(n) = F(n-1) + F(n-2)$ for $n > 1$
$F(0) = 0$ and $F(1) = 1$

Fibonacci
input : *a integer n*
output : *the Fibonacci number of n*

```
if n = 0
    f ⟵ 0
if n = 1
    f ⟵ 1
if n > 1
    p ⟵ Fibonacci(n − 1)
    q ⟵ Fibonacci(n − 2)
    f ⟵ p + q
return : f
```



Height? Cost?

~ 1

~ 2

~ 4

~ 6

~ 2

In each node in this tree, we have to perform a constant number of instructions.

1. How high is the tree? The height is n.
2. How many nodes (function calls) are in this tree? $< \sum_{i=0}^{n} 2^i = 2^{n+1} - 1 = O(2^n)$ upper bound
   $> \sum_{i=0}^{\frac{n-1}{2}} 2^i = 2^{\frac{n+1}{2}} - 1 = \Omega(2^{\frac{n}{2}})$ lower bound

Fibonacci Recursive

- Complexity F(n)?

  Exponential in n

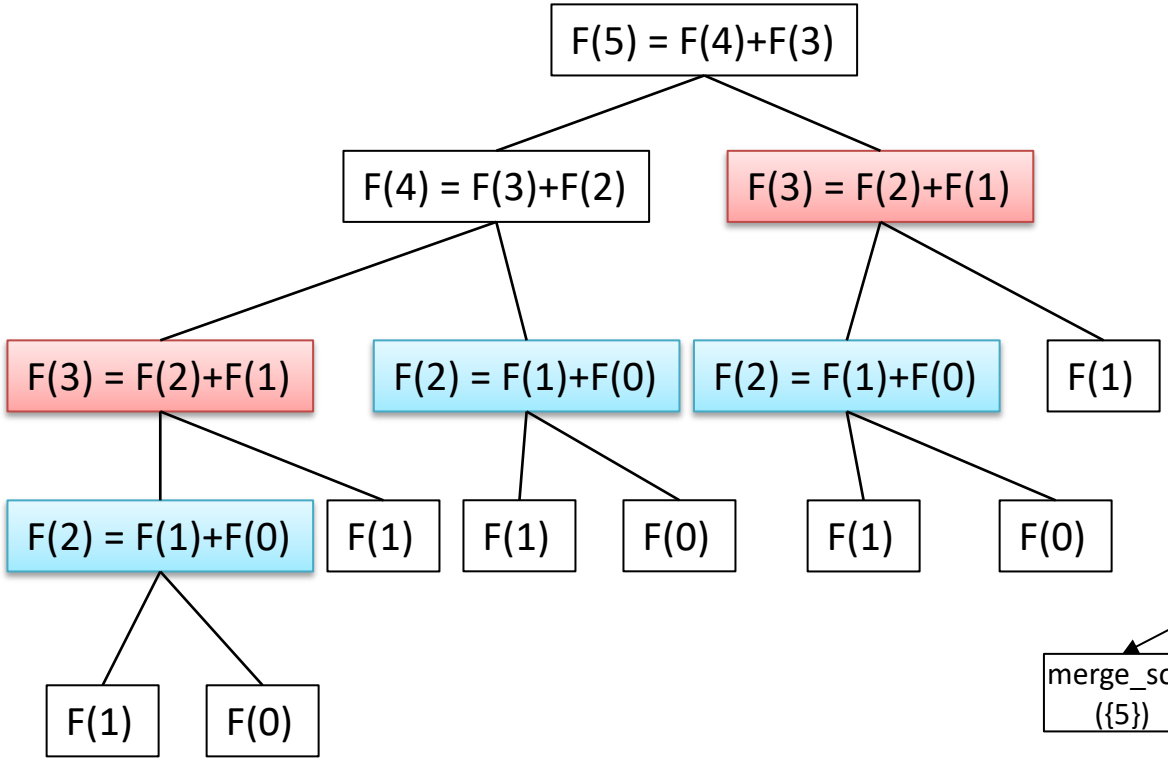
- Is there a better solution?

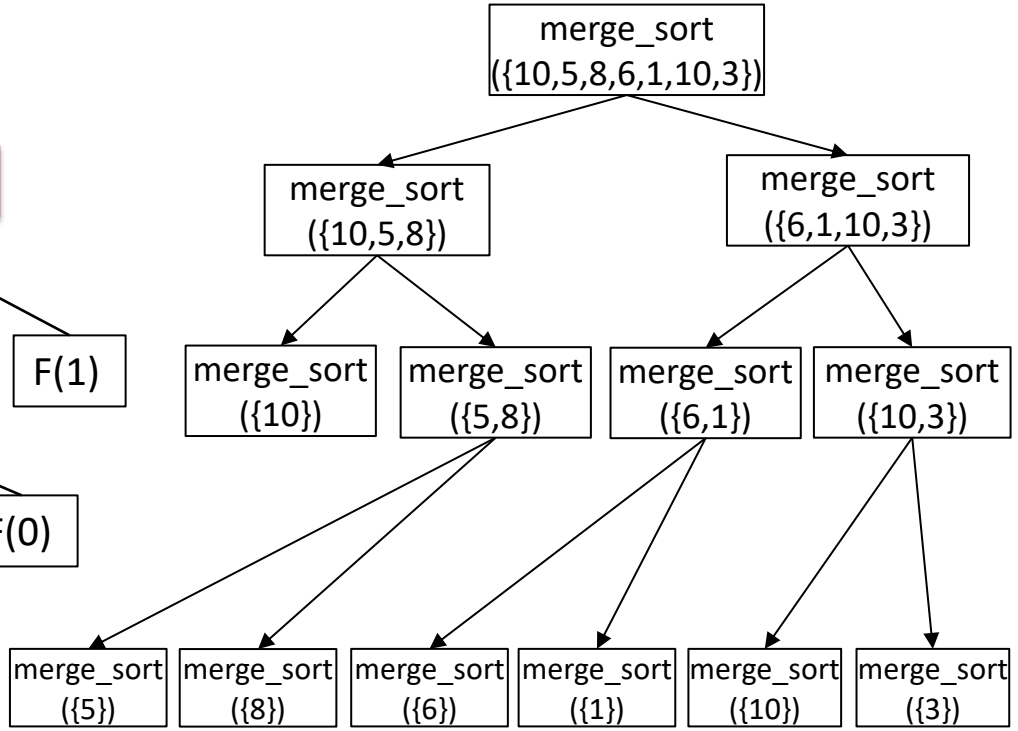
Idea: do not perform the same calculation many times but store the already calculated value for later use

Dynamic Programming

- **Dynamic programming** is a strategy to solve problems.
  - It applies to problems for which we can find an **optimal solution** by breaking it down into smaller **optimal overlapping** sub-problems in a recursive manner.
  - To solve such problems efficiently we **memorize** the solutions of sub-problems to avoid re-computation. (This technique is called **memoization.**)
- If the sub-problems are not overlapping the solving strategy is called ”**divide and conquer**” (e.g., in merge sort)

Overlapping vs Non Overlapping





Fibonacci with Memoization (Version 1)

- How to avoid recomputation?
- Idea: store all the computed results and check if the result has been already computed before computing it.

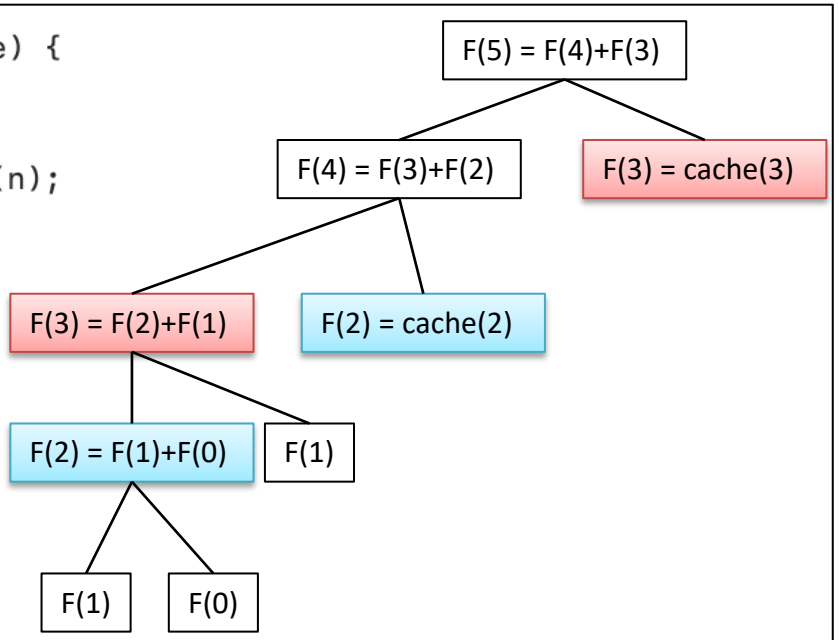
```
int fibonacci1 (int n, map<int, int>& cache) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (cache.contains(n)) return cache.at(n);

    int p = fibonacci1(n - 1, cache);
    int q = fibonacci1(n - 2, cache);
    cache.insert_or_assign(n, p + q);

    return p + q;
}

int main() {
    map<int,int> cache;
    cout << "Computing with memoization version 1: " << fibonacci1(45, cache) << endl;

}
```



Fibonacci with Memoization (Version 2)

Memorize the last two computations (x and y)

Fibonacci

input : *a integer* $n > 1$
output : *the Fibonacci number of* n

```
x ⟵ 0                    //f_{n-2}
y ⟵ 1                    //f_{n-1}
for i from 2 to n
    z ⟵ x + y            //f_n ⟵ f_{n-2} + f_{n-1}
    x ⟵ y                //f_{n-2} ⟵ f_{n-1}
    y ⟵ z                //f_{n-1} ⟵ f_n
return : z
```

Complexity?

Fibonacci

input : *a integer* $n > 1$
output : *the Fibonacci number of* n

```
x ⟵ 0                    //f_{n-2}
y ⟵ 1                    //f_{n-1}
for i from 2 to n
    z ⟵ x + y            //f_n ⟵ f_{n-2} + f_{n-1}
    x ⟵ y                //f_{n-2} ⟵ f_{n-1}
    y ⟵ z                //f_{n-1} ⟵ f_n
return : z
```

A. Θ(log n)

B. Θ(n)

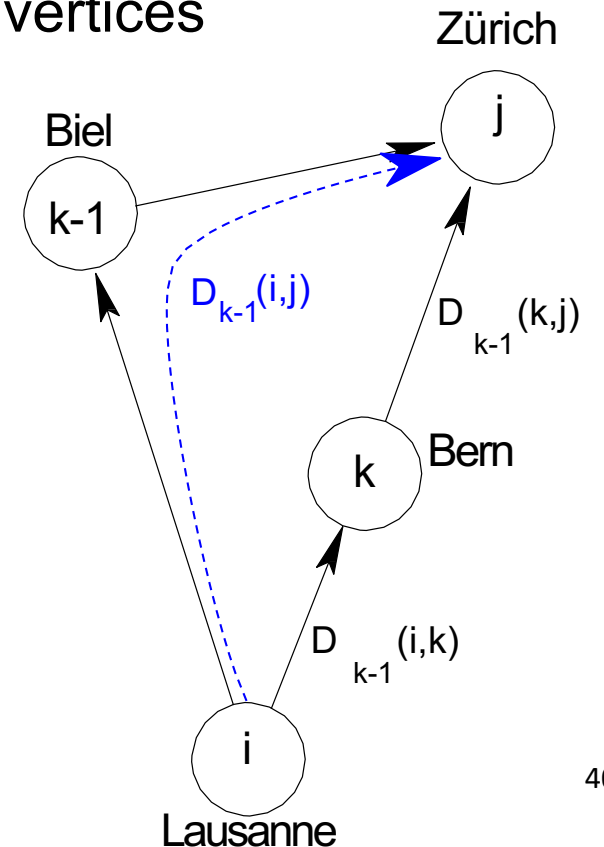
C. Θ(n^2)

D. Θ(2^n)

Dynamic Programming – Floyd

Computation of the **shortest path**, for example between all the train stations of the CFF network.

Given a graph G with vertices 1,2,..,N, consider a function called $D_k(i,j)$ that returns the shortest path from i to j using only vertices from 1 to k as **intermediate points**.



Key Idea of the algorithm:

The shortest path to go from i to j (e.g., Lausanne to Zürich) is the shortest path among:

1. The shortest known path from Lausanne to Zürich (using nodes 1 to k-1), and
2. The path from Lausanne to Zürich by traversing a city not known yet (e.g., Bern)

$$D_k(i, j) = \min \{D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)\}$$

Dynamic Programming – Floyd

ShortestPath

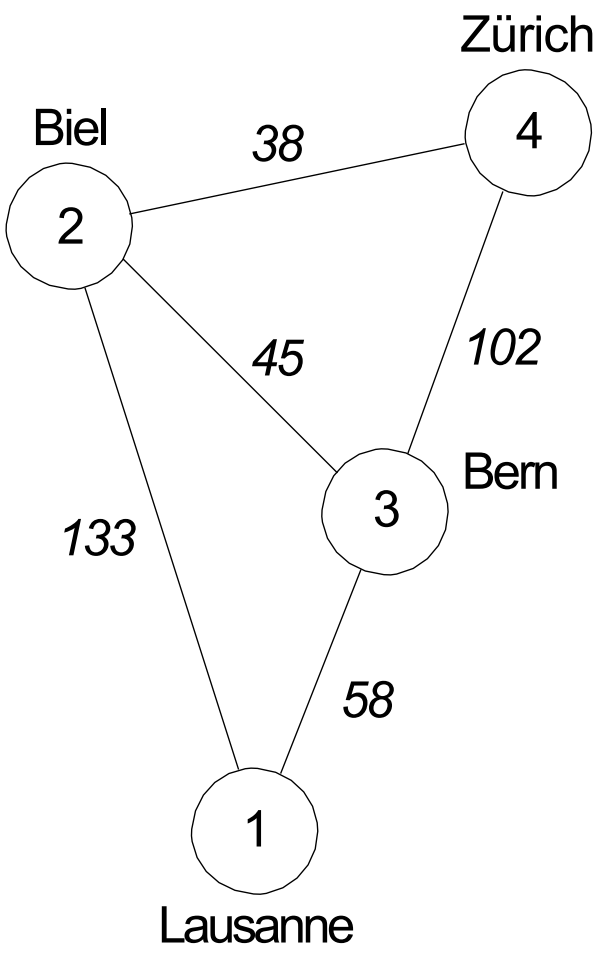
input : *a table* $C(i,j)$ *with the cost of taking the direct road between city* i *and* j *; the cost is* ∞ *if there is no direct road between city* i *and* j *and the cost is* 0 *if* $i = j$ *; there are a total of* n *cities*
output : *a table* $D(i,j)$ *with the shortest distance between* n *cities*

```
//Initialisation
for i from 1 to n
    for j from 1 to n
        D(i,j) ⟵ C(i,j)
//Update
for k from 1 to n
    for i from 1 to n
        for j from 1 to n
            if D(i,j) > D(i,k)+D(k,j)
                D(i,j) ⟵ D(i,k) + D(k,j)
return : D
```

i … the index of the source city
j… the index of the target city

k … the index of the city we allowed to visit (in addition to source and target city)

Floyd's Algorithm: Example



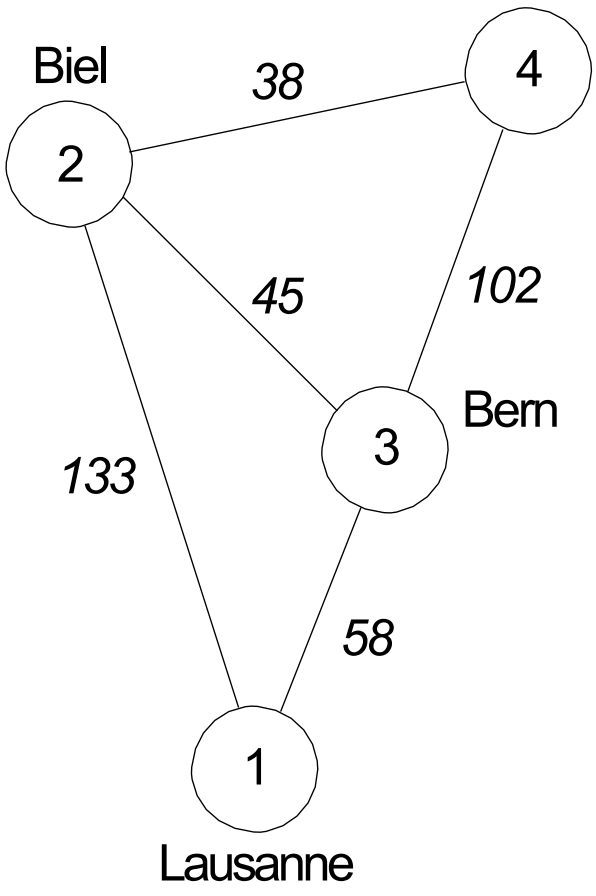
(fictitious data)

D(i,j)	Lausanne	Biel	Bern	Zürich
Lausanne	0	133	58	∞
Biel	133	0	45	38
Bern	58	45	0	102
Zürich	∞	38	102	0

k=1(Lausanne): Going throw Lausanne does not give any improvements.
k=2(Biel): Going throw Biel improves the distances

- from Lausanne to Zürich and
- from Zürich to Bern

Floyd's Algorithm: Example



(fictitious data)

$D_0 = D_1$

	Lausanne	*Biel*	*Bern*	*Zürich*
Lausanne	0	133	58	∞
Biel	133	0	45	38
Bern	58	45	0	102
Zürich	∞	38	102	0

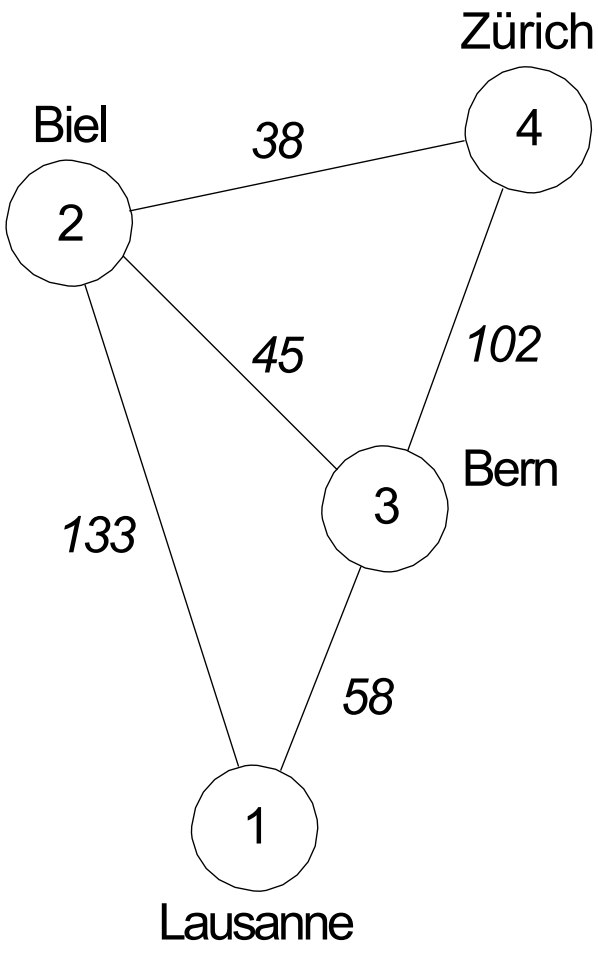
$D_2 =$

	Lausanne	*Biel*	*Bern*	*Zürich*
	0			
	133	0		
	58	45	0	
	171	38	83	0

k=1(Lausanne): Going throw Lausanne does not give any improvements.
k=2(Biel): Going throw Biel improves the distances

- from Lausanne to Zürich and
- from Zürich to Bern

Floyd's Algorithm: Example



(fictitious data)

D_2 =

Lausanne	Biel	Bern	Zürich
0			
133	0		
58	45	0	
171	38	83	0

D_3 =

Lausanne	Biel	Bern	Zürich
0			
103	0		
58	45	0	
141	38	83	0

k=3 (Bern): going throw Bern improves the distances

- from Lausanne to Biel and
- from Zürich to Lausanne

K=4 (Zürich): going throw Zürich does not give any improvements

Note: it also works for asymmetric graphs (directed graphs)

Complexity?

ShortestPath

input : *a table* $C(i,j)$ *with the cost of taking the direct road between city* i *and* j *; the cost is* ∞ *if there is no direct road between city* i *and* j *and the cost is* 0 *if* $i = j$ *; there are a total of* n *cities*
output : *a table* $D(i,j)$ *with the shortest distance between* n *cities*

```
//Initialisation
for i from 1 to n
    for j from 1 to n
        D(i,j) ⟵ C(i,j)
//Update
for k from 1 to n
    for i from 1 to n
        for j from 1 to n
            if D(i,j) > D(i,k)+D(k,j)
                D(i,j) ⟵ D(i,k) + D(k,j)
return : D
```

A. $\Theta(n^2)$
B. $\Theta(n^3)$
C. $\Theta(n^4)$
D. $\Theta(n^5)$

Conclusion

- There is no miracle recipe to find an algorithm, but there exist big families of resolution strategies:
  - decompose (e.g., “recursion”): try to solve the problem by decomposing it in simpler (or smaller) instances
  - decompose and regroup (e.g., “dynamic programming”): memorize intermediate computations to avoid executing them several times