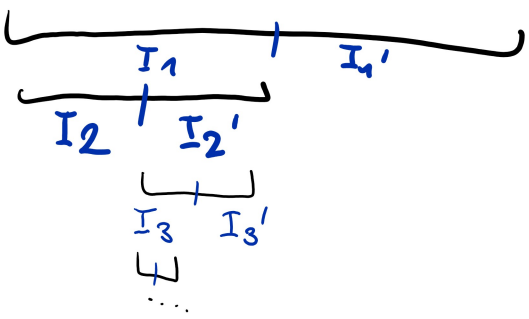
Divide & Conquer

1. Structure

- **Partition**

  divide (often by 2) espace de recherche at each step to (in the optimal case) only consider one of the subspace.

  i.e.



  Until we get a subproblem of the min size poss. to use "the trivial approach" on it.

  i.e. $\xrightarrow[m \text{ times}}$ until I_{m+1} / I_{m+1}' are trivial subpbms

- **Trivial Resolution**

  ↳ algo to solve I_{m+1}. i.e. local solution to each triv. subp.

_ Combine:

Combine subproblems to obtain / infer global solutions.

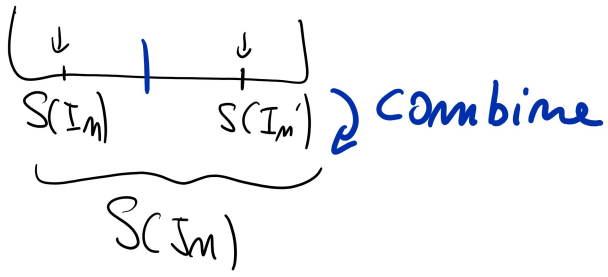
ie. let $S(I_m)$ be the solution of subp over I_m then:

$$\text{Combine}\left(S(I_m),\ S(I_m')\right) = S(J_m)$$

where $J_m = I_m \cup I_m'$

(There can be several ie. $J_{m_1}, J_{m_2}, \ldots$, see ex below)

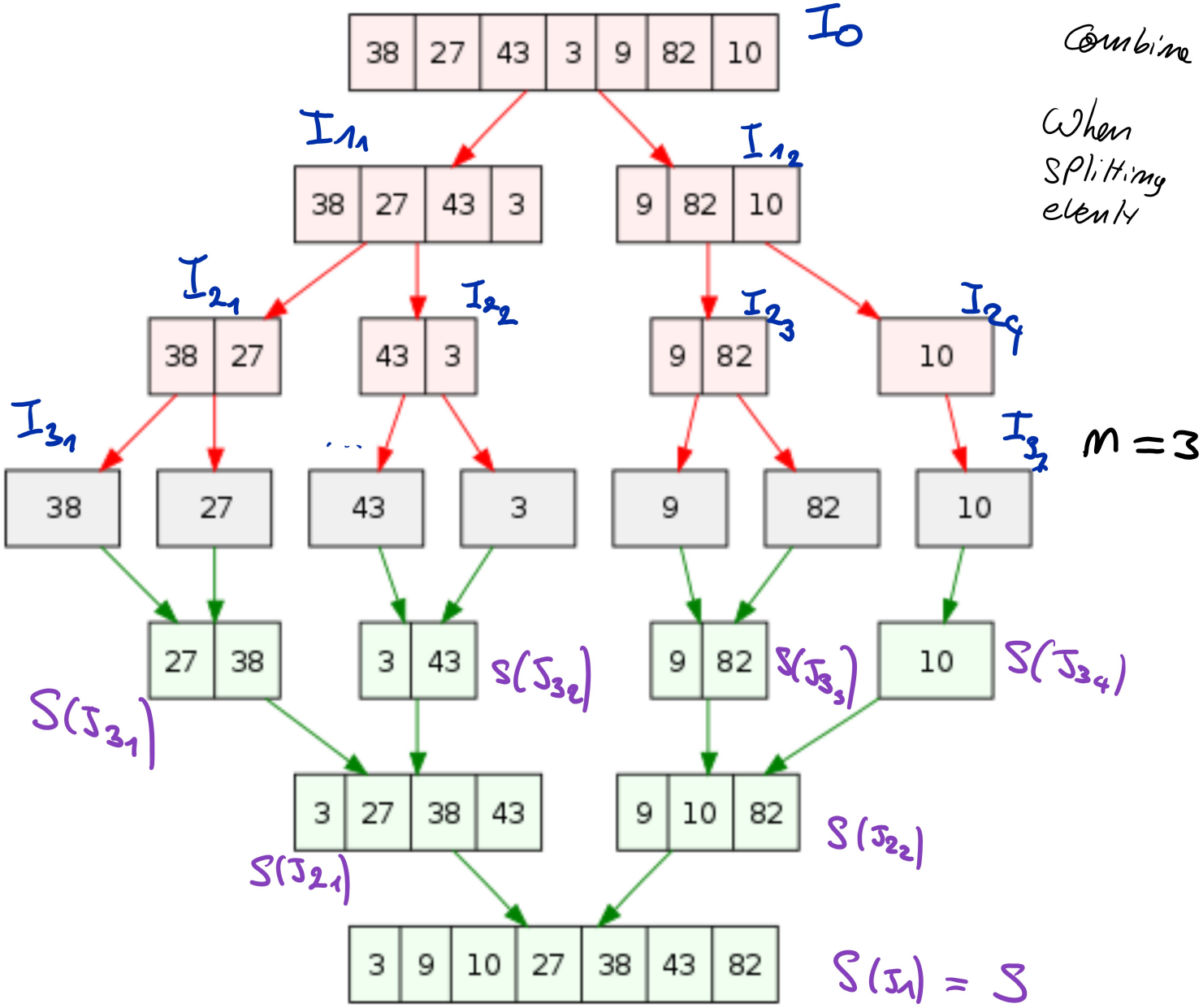
i.e. $S(J_1)$ = global Solution



1. Partition : $I_k \mapsto I_{k+1},\ I_{k+1}'$
2. triv Res : $I_m \mapsto S(I_m)$
3. Combine : $S(I_m), S(I_m') \mapsto S(J_m)$

$\lceil \log_2(7) \rceil = 3$ so 3 steps to partition and 3 to combine (group of 2)

When splitting evenly



The name "divide and conquer" is sometimes applied to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list (or its analog in numerical computing, the bisection algorithm for root finding).[2] These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use tail recursion, they can be converted into simple loops.

Always watch out & make sure to recognize these types of problems.

Algorithm efficiency [edit]

The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms.

In all these examples, the D&C approach led to an improvement in the asymptotic cost of the solution. For example, if (a) the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problem's size n, and (b) there is a bounded number p of sub-problems of size ~ n/p at each stage, then the cost of the divide-and-conquer algorithm will be $O(n \log_p n)$.

Merge of Merge Sort:

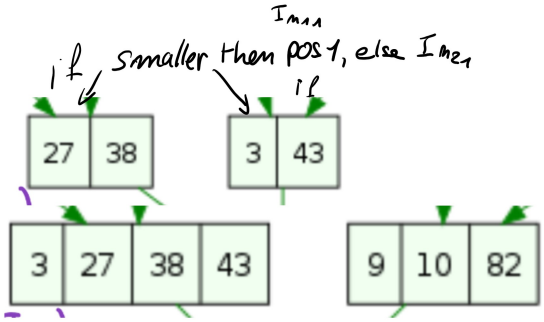
we are at bottom of rec calls (not seen here) when pop rec call stack:
end of loop → length of L, R → x2

```
# Copy data to temp arrays L[] and R[]
while i < len(L) and j < len(R):
    if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1

# Checking if any element was left
while i < len(L):
    arr[k] = L[i]
    i += 1
    k += 1

while j < len(R):
    arr[k] = R[j]
    j += 1
    k += 1
```

Store smallest and increase idx of array that contained it
(since sorted if L[i] ≤ R[j] then L[i] ≤ R[j+x] ∀x)
↪ hence incr i



☆ Basic Structure/Solution for this kind of pblm

Soit Small la f qui définit si un sous pb est trivial, G, celle qui résout ces pb triv. partition, combine (assez explicite) et P celle qui va nous indiquer avec quel sous-pb continuer

```
dc(I):
    dc_rec(I_m, m, N):
        if Small(I_m, m, N): return G(I_m, m, N)
        k = partition(I_m, m, N)
        return dc_rec(Im[:k], m+1, N) if P(Im, m, N) else
                                    dc_rec(Im[k:], m+1, N)

    return dc_rec(I, 0, len(I))
```

(I.e. pas besoin de combine → résolution que 1 seul sub prob)

☆ Résolution (rec) cas général

Soit small, partition ... définis comme ci-dessus.
à l'intérieur de dc (i.e. accès à nums)
- "formelle"

```
def dc(nums):
    def G(p,q): ...                          # forcément => dépend impl
    def combine(p,q):...                     # soi intermed => pas r p,q idx
    def partition(p,q):...
    def dc_rec(p, q):
        if small(p, q): return G(p,q)
        else: // else is useless due to this return
            m = partition(nums, p, q)
            return combine(dc_rec(p, m), dc_rec(m+1, q)) *
    return dc_rec(0, len(nums)-1)
```

* PS : Si on passe des slices d'array, on doit passer (nums[p:m+1], nums[m+1:q+1])
car nums[m] doit faire partie de la partie de gauche
et nums[q] doit faire partie de la partie de droite.

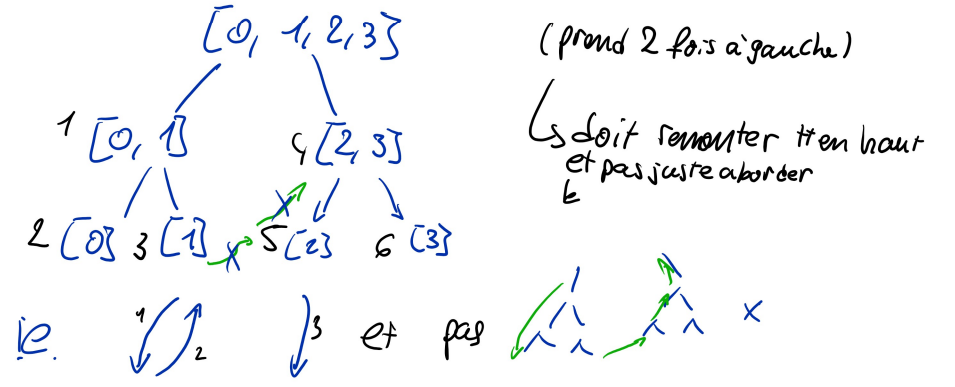
- cas "officieux"

```
def dc(nums:List[int]):
    if not nums or len(nums)==0: return <invalid value>
    if len(nums)==1: return G(0,0)  # or (0,1) depends

    def combine(S_i1, S_i2):...
    def G(p:int, q:int): ...

    def dc_rec(p:int, q:int):
        if p≥q: return G(p,q)  # trivial case (often for nums[p])
        m = (p+q)//2           # m := ⌊(p+q)/2⌋
        return combine(dc_rec(p,m), dc_rec(m+1,q))
```

Si c'est bien fait les rec calls vont "remonter toute la pile de gauche" avant d'attaquer celle de droite ie. mettons qu'on ait (grâce au return)



(prend 2 fois à gauche)
↳ doit remonter tt en haut et pas juste aborder

Exemple trivial sum array:

```
          [0, 1, 2, 3, 4, 5]
p0 = 0  m0 = 2  q0 = 5 [0, 1, 2] | [3, 4, 5]
p0 = 0  m0 = 1  q0 = 2 [0, 1] | [2]
p0 = 0  m0 = 0  q0 = 1 [0] | [1]
nums[0] = 0

nums[1] = 1

nums[2] = 2

p0 = 3  m0 = 4  q0 = 5 [3, 4] | [5]
p0 = 3  m0 = 3  q0 = 4 [3] | [4]
nums[3] = 3

nums[4] = 4

nums[5] = 5

expected: 15, obtained: 15
```

recommence du haut ([3,4,5])

ici:

```
G(p, q):
    return nums[p]
```

et

```
combine(S_i1, S_i2):
    return S_i1 + S_i2
```

☆ Si l'algo se résout "in place"

Dans ce cas là (i.e. on a pas besoin de return)
On a pas de tail rec ni rien on a juste

P.ex. pr le quick sort:

```
def quicksort(A)
    def qs_rec(p, q):
        if p >= q: return
        m = partition(p, q)
        qs_rec(p, m)
        qs_rec(m+1, q)
    qs_rec(0, len(A)-1)
```

Example merge sort:

```
def mergeSort(arr):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # Into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

    # Checking if any element was left
    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
```

Backtracking

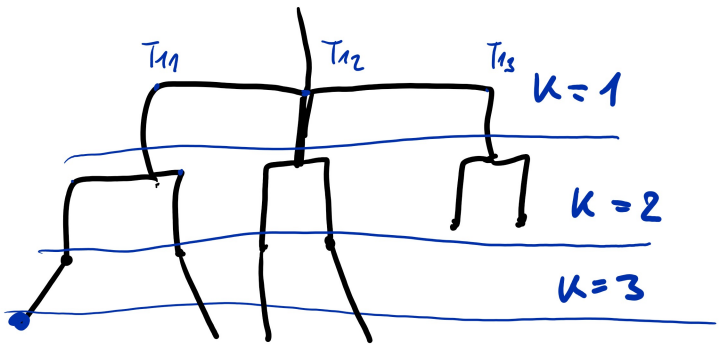
1. Structure

- Itérer (à l'étape k) sur l'ens. des Val. courantes possibles (déf par les cnds explicites T_k)

- Tester si $(x_i)_{i=0}^{k}$ est une solution locale (B_k) (ie. si c'est un candidat potentiel de sol glob.)
  - Si oui: Tester si c'est une sol glob (P)
    (⚠ il peut: $\exists$ Sol glob pour $k < N$)
    - ↳ si oui: Sol trouvée
    - ↳ Si non: passer au Step $k+1$
      (fait naturellement un arbre, k = depth)

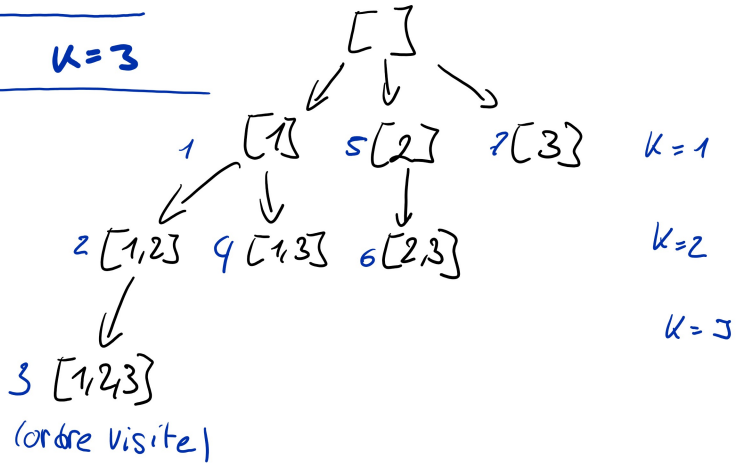
  - Si non: continuer l'itération de la Step k

- Une fois l'itération Step k finie, rec call se fini et remonte (backtrack) à celle (en cours) de la step $k-1$



N = 3 (ici)

e.g. powerset de [1,2,3]



(ordre visite)

2. Résolution cas général

```
def BT(...):
    def T(X, k, N):          # déterminé d'après X[0:k]
    def B(X, k, N):          # déterminé d'après X[0:k+1]
    def P(X, k, N):          # déterminé d'après X[0:k+1]
    out = None
    def rBT(X, k, N):
        nonlocal out
        for y in T(X, k, N):
            X[k] = y
            if B(X, k, N):
                if P(X, k, N):
                    out = X[:k+1]      # return solat {X[i] for i in range(k+1)}
                    return out         # exit point of rec call. 1st stmt after exit
                rBT(X, k+1, N)         # as soon as out isn't None : direct returned
                                       #   ↳ break for & call stack
                if out is not None: return out

    return rBT([None]*N, 0, N)
```

NB : pour return toutes les solutions :
enlever les return out / if out is not ... et
remplacer par "out.append(X[:k+1])". Ici out sera

Dynamic Programming

1_ Intuition

- Overlapping Subproblems
- Optimal substracture
- Induction

2_ "Structure"

1. We solve smallest (trivial) subproblem optimally "k=0" (basis step)
2. Then, by going through (all) the $(i \in [0, k])$ previous solutions and analyzing them, find how we can guarantee building the optimal solution for step k+1 "$k \xrightarrow{?} k+1$" (Induction Hypothesis)
3. With that relation, implement finding the optimal solution for any step k+1, by recursively resolving and storing the answer to subprob $[0 \ldots k]$ into a matrix. (Inductive step)

"$\forall k \geq 0: s(k) \rightarrow s(k+1)$"
"$\forall k \geq 0: s(k)$"

Example : "All pairs shortest path problem" (Floyd Algo)

Given a DWG (directed weighted Graph)
find shortest path between every vertices (v_i, v_j) in G.

$G = (V, E)$ $\quad V = [\![1, \underbrace{n}_{=|V|}]\!]$ $\quad L \in \mathbb{M}_{n \times n}$, $L(i,j) = \begin{cases} 0, & i=j \\ \infty, & \text{no edge} \\ \omega(i,i), & \text{otherwise} \end{cases}$

$\omega(i,j) :=$ weight of edge (i,j).

$D \in \mathbb{M}_{n \times n}$, $D(i,j)$: shortest path from v_i to v_j

things to check:

- Opti Sub prop. holds? (ie. if X is $\overset{\text{opti}}{\text{sol}}$ for k, then X holds the opti sol for all sub $0 \dots k-1$)

  ↳ Assume p is the shortest path from v_i to v_j, (ie. p is optimal $= D(i,j)$)

  if $v_k \in p \rightarrow (v_i, \dots v_k)$ is optimal $\wedge$
  $(v_k, \dots, v_j)$ is optimal $\quad (p = (v_i, \dots, v_k, \dots, v_p))$

  ($\exists$ shorter path from $v_i \rightarrow v_k$ ie.
  $D(i,k) \neq (v_i, \dots, v_k) \rightarrow \exists p', v_k \notin p' \wedge \underbrace{p' = D(i,j) < p}_{\text{contradiction}}$)

- Triv resol? ("$k=0$") Basis Step

(B.S.) Here we can just initialize D to L since the B.S. will be all the path for all adjacent nodes (ie. if $(i) \xrightarrow{x} (j)$ then the shortest path is just "given by the edge" (length 1, $\omega = x$)).

↳ "skipped"

Approach:

(I.H) _ Assuming we filled

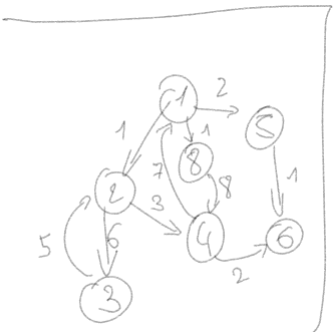
find a recurrence relation to compute (Strong induction)

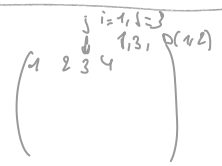
here a step k doesn't rep state of an int but a matrix D

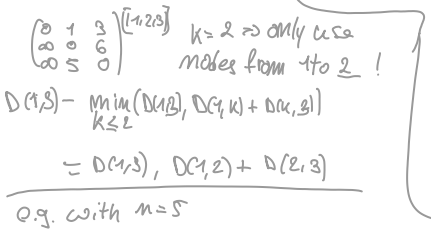
ie. $D_0, D_1 \dots$ between 2 steps we iterate through whole matrix.

And at Step k, we stored the optimal path from i to j

(for all pairs (i,j) i.e. whole mat) using only nodes from [1 to k].







ie. at Step k+1 we check for all paths from i, to j

if its shorter to pass by k+1 instead. i.e. if the path from

i to k+1, and from k+1 to j is shorter than the current

one from i to j.

We don't have the pblm "where to put the stop to k+1" bc we rec store the crt best path from $i \to k+1$ and $k+1 \to j$ and we completely replace former by this one

s.p between i,j at Step k is min length of all path

Starting from i and ending in j, using only nodes 1 to k.

Let S_{ij} the final optimal path from i to j

At each Step k (for whole mat ie. $\forall (i,j) \in N^2$):

- either $v_k \notin$ optimal path S_{ij} : change nothing in D(i,j)
- $v_k \in S_{ij}$: $D[i,j] = D[i,k] + D[k,j]$

So instead of implementing smth to check wether $v_k \in S_{ij}$ we just take $\min(D[i,j], D[i,k] + D[k,j])$ and repeat for all k. (So also for all (i,j))

The rec relation becomes:

$$\begin{cases} D_{k+1}(i,j) = \min_{\forall (i,j) \in [\![0, n-1]\!]^2} (D_k[i,j], D_k[i,k+1] + D_k[k+1,j]) \\ D_0 = L \in \mathbb{M}_n \end{cases}$$

$\Rightarrow$ Exactement comme le COINS problem:

Comment obtenir le path qu'on veut

en combinant, seulement tous les opti paths qu'on a déjà trouvé ? => Au début seulement avec les edges $(x, x+1)$

est-ce que faire $x \to k \to x+1$ plus court que $x \to x+1$!

puis on a des paths de + en + long et on peut limiter $k-1$ et restart

It's not bc D(x,y) contains a value that its the s.p. it is only if k=n ie. end of algo (recursion. D(x,y) can contain "temporary best" value that will get corrected later

Because the S.p. between i,j is fully correct only at step k=n,
all the paths between pairs are built (Checked for optimality) gradualy
ie. up to step k for each k.
Meaning the cost $D_k(i,j)$ is not optimal for $k < n$ but the path being
constructed in P (array of S.p.) ~~is optimal~~! (up to k). (no thats why we iterate through whole mat at each step)
The "real" indaction is on the paths of length k being constructed
at the same "level" for each (i, j). And since the $P_k(i,j)$ isn't complete then
its cost $D(i,j)$ is wrong,

At each step k we traverse the entire matrix to see if
passing by k on the road from i to j is shorter (for all i,j bc whole mat)

Shortest path between 1, k is D(1, k), the s.p. (short. path.)

between (k, k+1) is either (k, k+1) (if it exists and is the min)

or $\min_{\forall j}(d(k,j) + d(j, k+1))$, i.e. the min of the length of all path

starting from k and ending to k+1. (i.e. trying all the

intermediary steps possible and taking the best option after having

seen all of them).

B&B

Structure

1 : listOfChildren (getChildren) (node : Node)
→ Set

2 : cost (ĉ) ⚠ Pas de Node en arg!
(va être appelée ds constructeur)
⇒ (val, path, cost, parent) (signature)

3 : addToLiveNodes (maintain PQ structure)
ie. sorted

4 : nextENode : just pop

5 : P (defines if a Node is a solution)

2.1 fonction de coût optimale?

pour qu'elle soit optimal ⇒ 4 prémices doivent être ok.

ie pour une liste des live nodes L et x^* une sol :
on doit avoir $\hat{C}(x^*) = C(x^*)$ et $\hat{C}(x) \leq C(x)$

Comme ça on obtient que si on maintient une priority queue sur L, l'E-node choisit a toujours le plus petit coût de L i.e.

si x^* est choisi alors

$$\hat{C}(x^*) = C(x^*) \leq \hat{C}(L) \leq C(L)$$

x^* est bien de coût min. ie est la solution optimal, (par def de C, C(root) = c(optimum))

ici $c(root) = \sum_{j=0}^{n} \min_{x \in I_j}(C(x))$, on voit bien que la somme des coût min est bien le min des coût (par déf du min).

ici C est donné explicitement, on peut juste prendre $\hat{C} = C$. → C prend pas en compte la contrainte ⇒ si $\hat{C}$ le fait