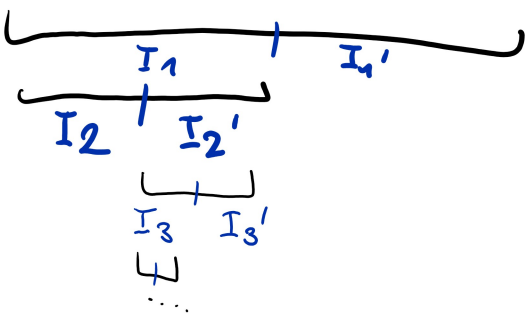
Divide & Conquer

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.

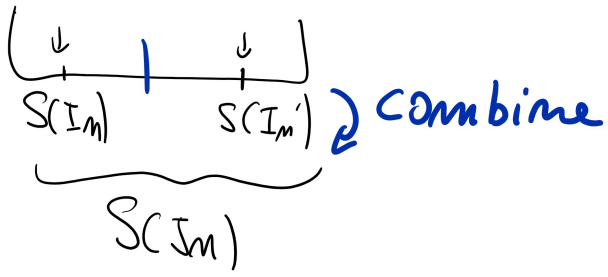
ie. let $S(I_m)$ be the solution of subp over I_m then:

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

where $J_m = I_m \cup I_m'$

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

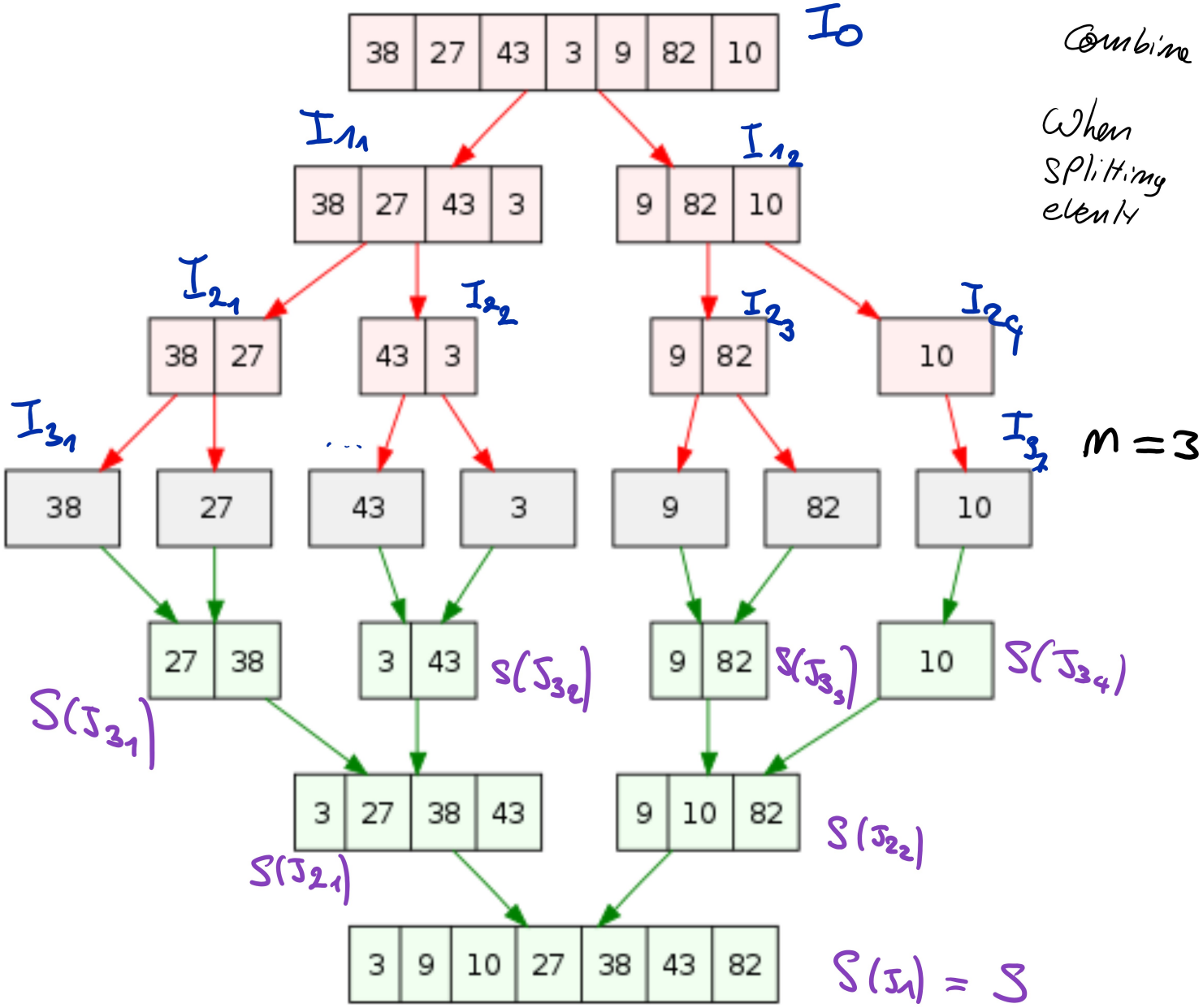
i.e. $S(J_1)$ = global Solution



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:

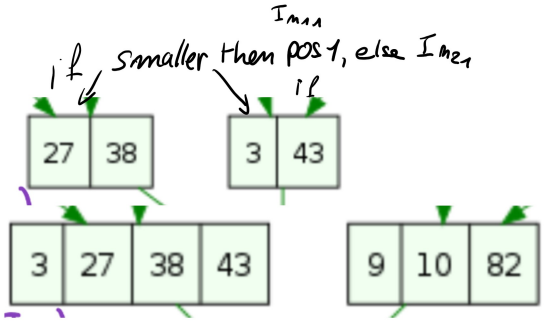
we are at bottom of rec calls (not seen here) when pop rec call stack:
end of loop → length of L, R → x2

```
# 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.

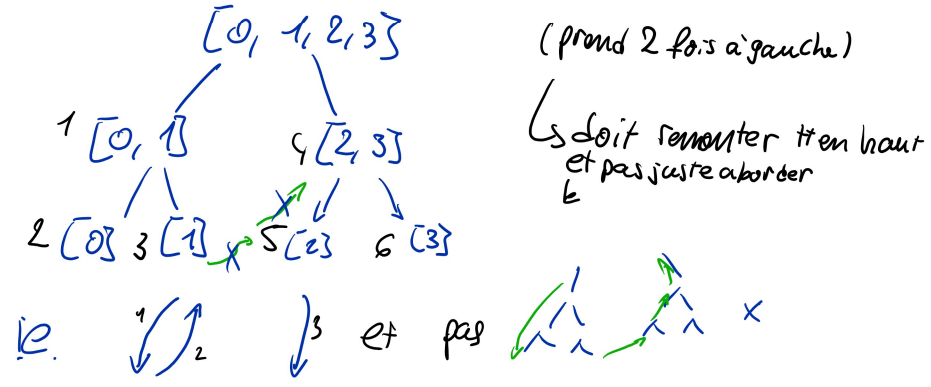
- cas "officieux"

```
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
```