

**Même si beaucoup de problèmes sont NP-difficiles,
tout N'est pas Perdu...**



Information, Calcul et Communication

Le problème du sac à dos

Olivier Lévêque

Le problème du sac à dos

- n objets, chacun pesant un certain poids
- un sac à dos de capacité maximale C
- Comment remplir au mieux le sac à dos?



Le problème du sac à dos

« Etant donné une liste L de n nombres entiers positifs et $C > 0$, trouver un sous-ensemble $S \subset \{1, \dots, n\}$ tel que

$$\sum_{i \in S} L(i) \leq C$$

et

$\sum_{i \in S} L(i)$ soit maximale.»

Hypothèses additionnelles:

- $L(i) \leq C$ pour toute valeur de i
- $\sum_{i=1}^n L(i) > C$



Le problème du sac à dos

= problème d'optimisation discrète (avec contrainte)

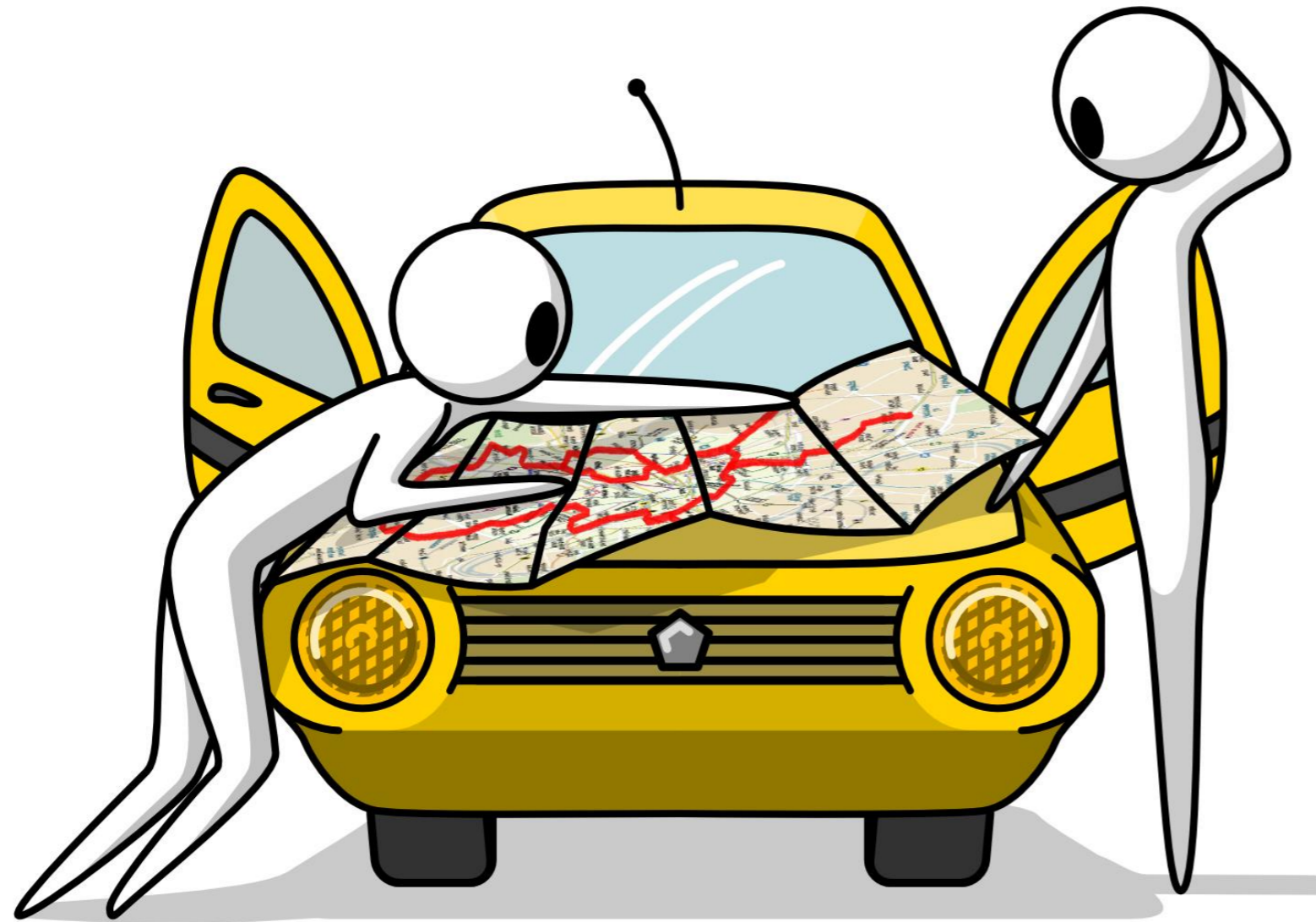
On ne sait pas s'il fait partie de la classe P, ni s'il fait partie de la classe NP.

Voici toutefois un algorithme polynomial en n qui remplit au moins la moitié du sac à dos :

1. Ordonner la liste L dans l'ordre décroissant
2. Chercher le nombre $k \in \{1, \dots, n - 1\}$ tel que

$$\sum_{i=1}^k L(i) \leq C \quad \text{et} \quad \sum_{i=1}^{k+1} L(i) > C$$

Alors $\sum_{i=1}^k L(i) \geq C/2$



Information, Calcul et Communication

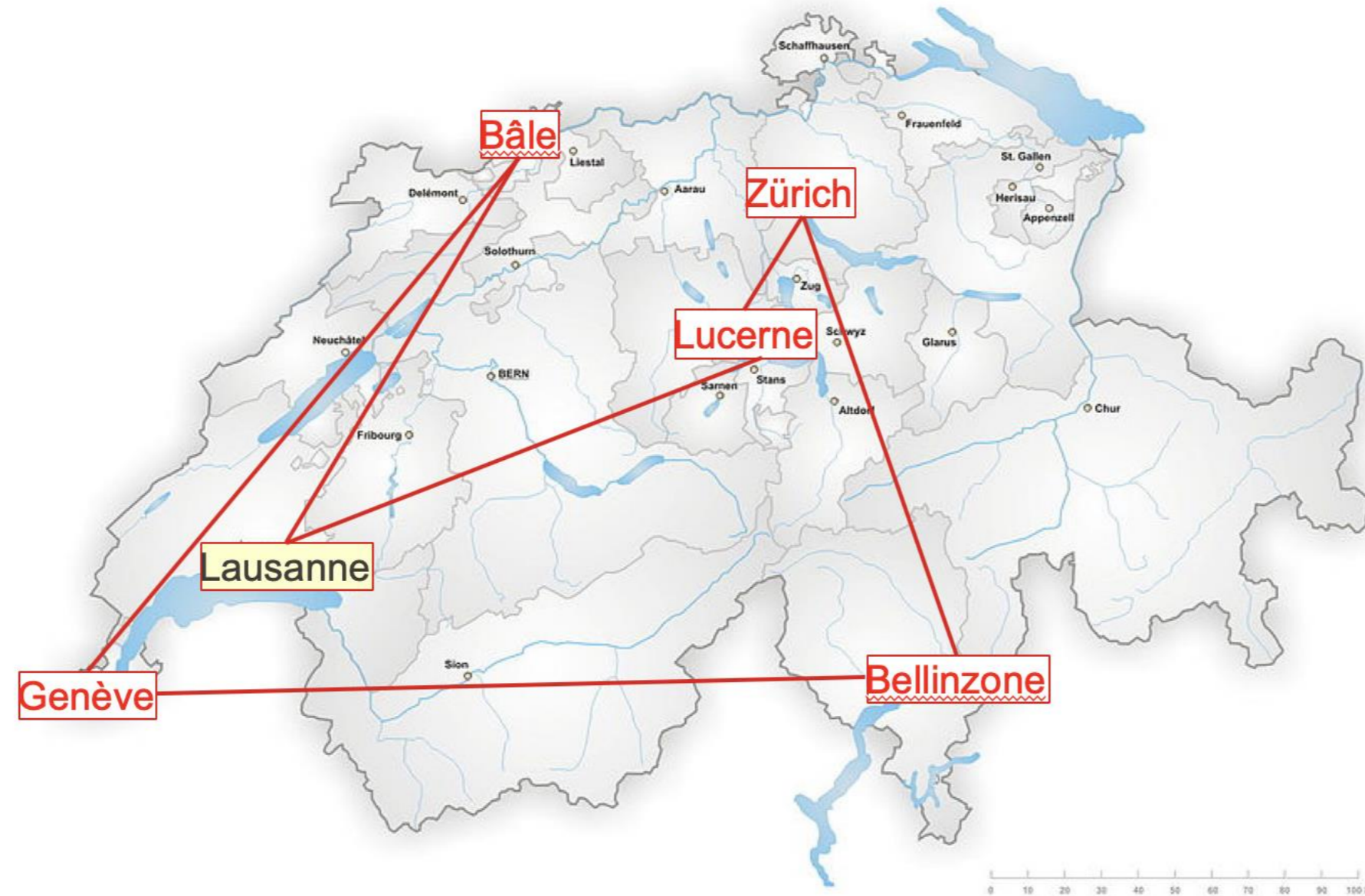
Le problème
du voyageur de commerce

Olivier Lévêque

Le problème du voyageur de commerce

Version « euclidienne » :

Etant donné un ensemble de n villes sur une carte, trouver le chemin fermé le plus court passant une et une seule fois par chacune de ces villes.



Note : On suppose ici des voyages à vol d'oiseau entre chaque ville.

Premier essai :

Tester tous les chemins fermés possibles

→ Cet algorithme trouve le chemin fermé optimal ✓

... mais les chemins fermés sont au nombre de $n! = n(n - 1)(n - 2) \dots 1$
impossible à mettre en pratique !

Deuxième essai :

1. Partir d'une ville choisie au hasard
2. Tant qu'il reste une ville non-explorée, rejoindre la ville non-explorée la plus proche
3. Quand toutes les villes ont été explorées, revenir à la ville de départ

→ complexité temporelle polynomiale en n ✓

... mais le chemin fermé résultant est loin d'être optimal en général !

Troisième essai :

1. Choisir un **chemin fermé** ($v_1, v_2, \dots, v_n, v_{n+1} = v_1$) au hasard
 2. Effectuer une boucle pour i allant de 1 à n :
 - Permuter les villes v_i et v_{i+1} dans le chemin si cette permutation raccourcit la longueur totale du chemin
- complexité temporelle polynomiale en n ✓
- ... meilleure performance moyenne que l'algorithme précédent, mais résultat toujours aléatoire

Algorithme de résolution avec garantie d'approximation (et complexité temporelle polynomiale en n)

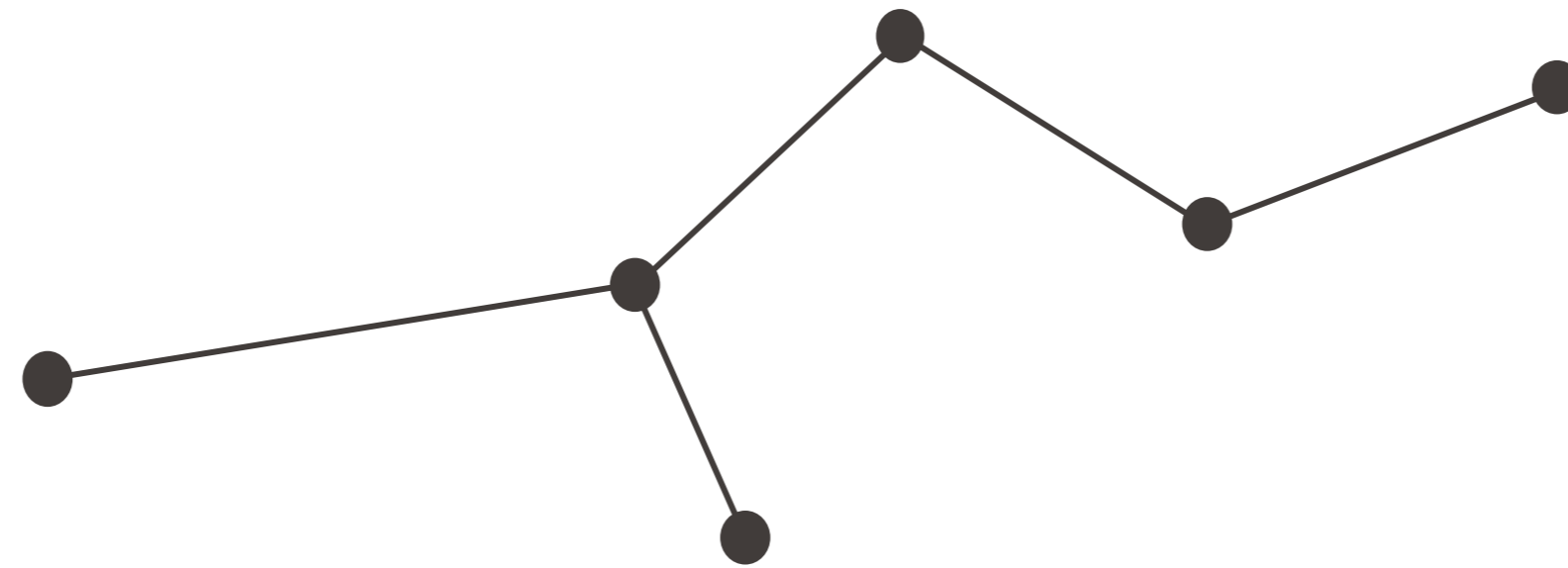
Théorème

Soit L_{min} la longueur du chemin fermé optimal (i.e., le plus court).
Il existe un algorithme de complexité temporelle polynomiale en n
permettant de trouver un chemin fermé de longueur $L \leq 2L_{min}$.

Première étape : arbre couvrant minimal

Etant données des villes sur une carte, on cherche à les relier par un arbre dont la somme des longueurs des branches soit minimale.

Exemple :



Il est possible de trouver un tel arbre (T) en temps polynomial en n :

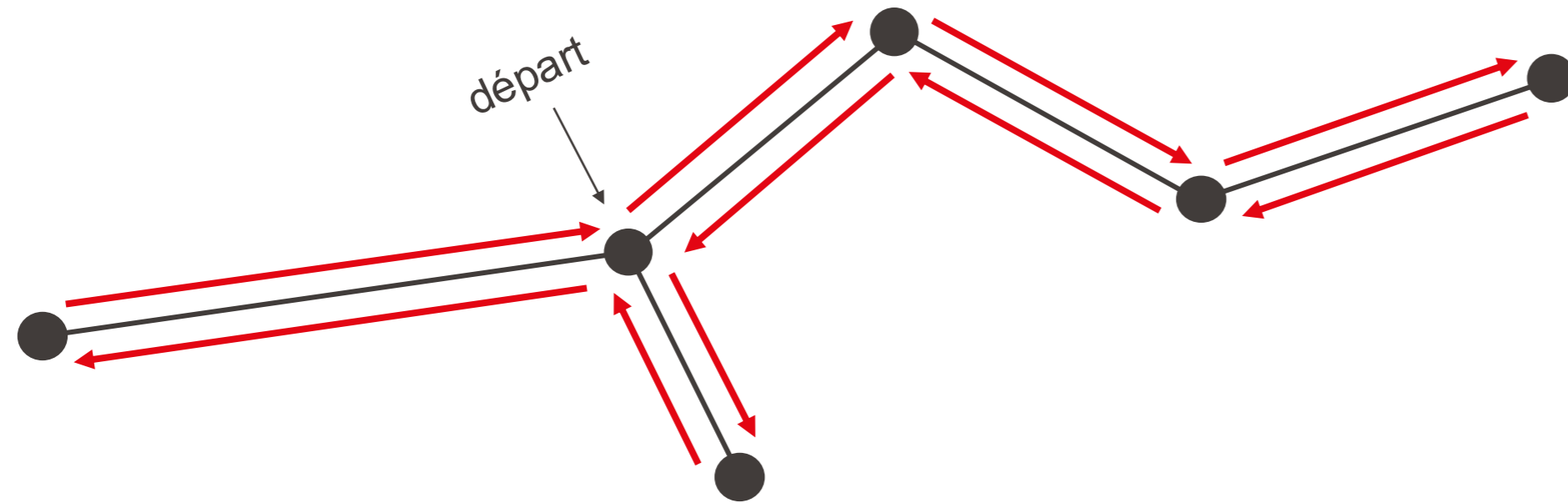
1. Commencer avec $T = \{\text{une ville au hasard}\}$ (= racine de l'arbre)
2. Chercher parmi les villes restantes la ville v qui soit la plus proche d'une des villes $w \in T$: rajouter v et la branche $v - w$ à T
3. Recommencer en 2. jusqu'à ce que T contienne toutes les villes

Remarque importante

Si L_T désigne la somme des longueurs des branches de l'arbre couvrant minimal, et L_{min} désigne la longueur du chemin fermé optimal qui passe une fois par chaque ville, alors $L_{min} \geq L_T$.

Deuxième étape : parcours le long de l'arbre

Illustration



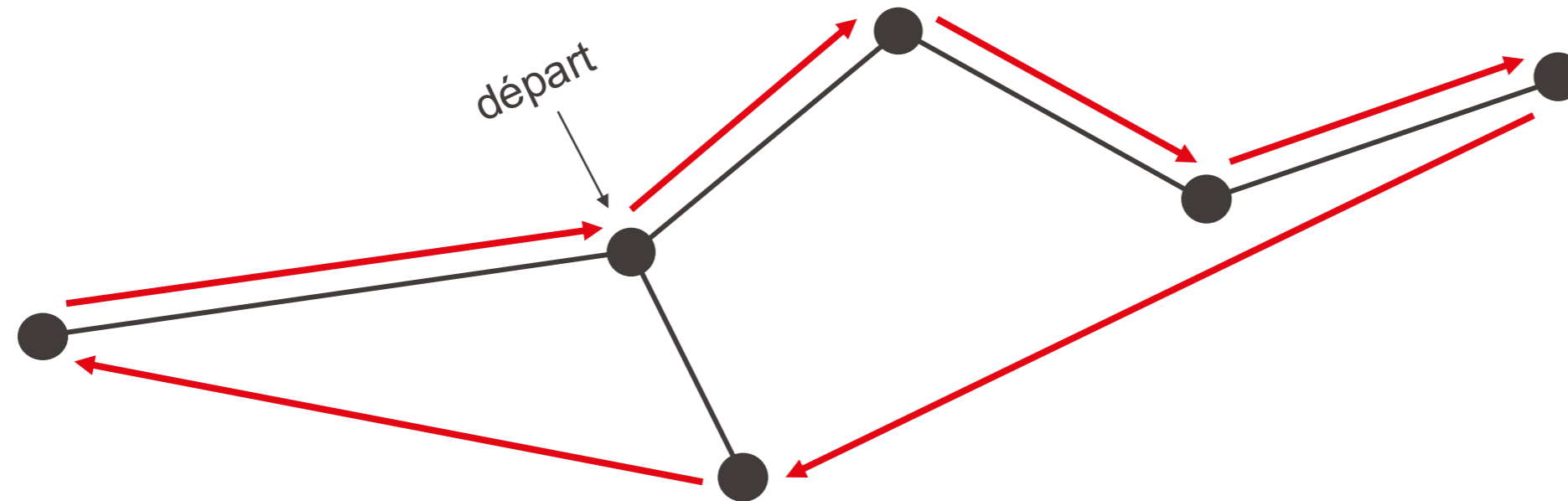
La longueur du chemin rouge vaut $2L_T \leq 2L_{min}$

Presque ce qu'on veut, mais le chemin rouge n'est pas un chemin qui passe une et une seule fois par chaque ville.

Troisième et dernière étape : prendre des raccourcis

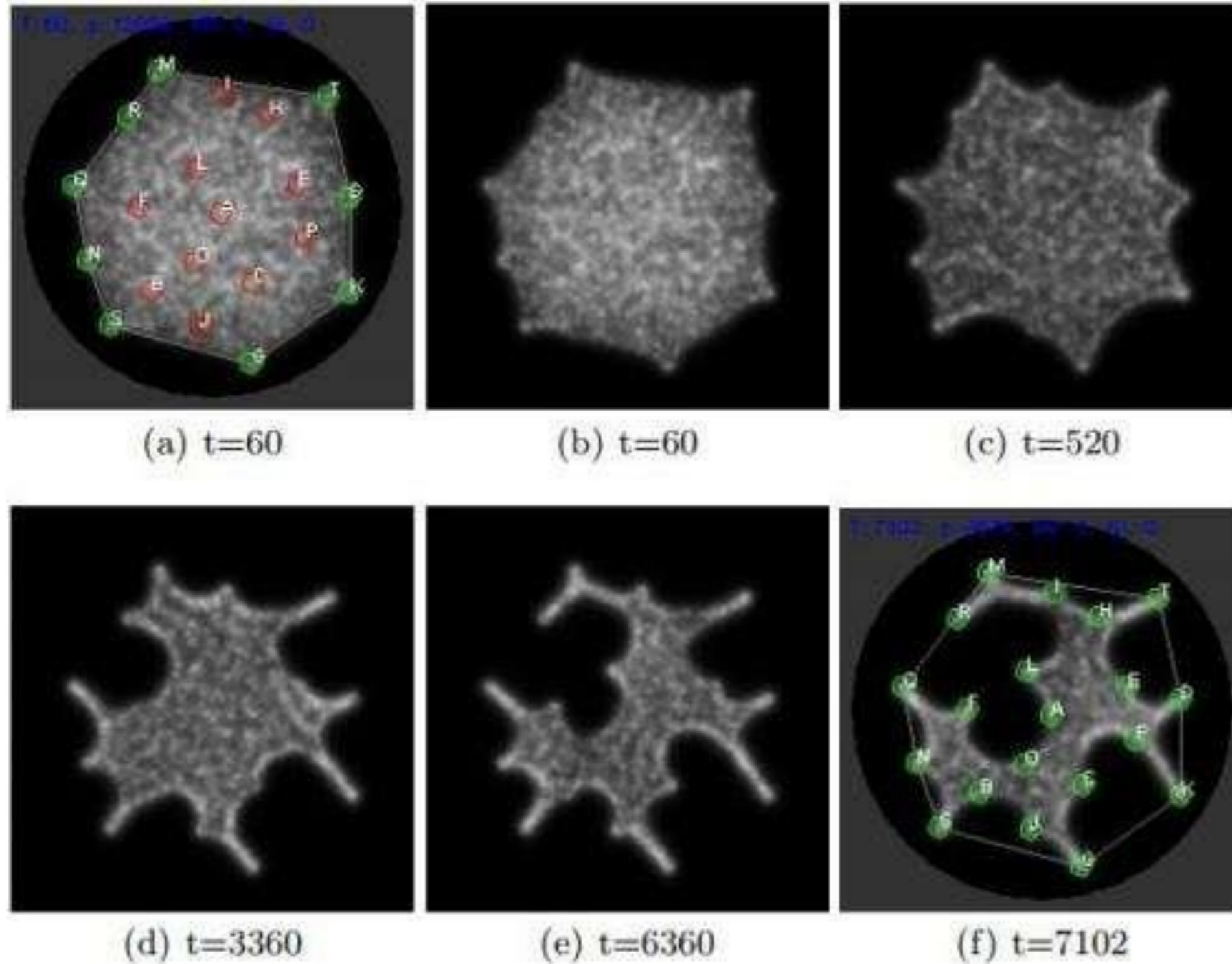
On modifie le chemin rouge en suivant la règle : dès qu'une ville a déjà été visitée, on prend un raccourci vers la ville suivante dans l'arbre.

Illustration



Ce nouveau chemin rouge n'est pas plus long que le précédent (donc $\leq 2L_{min}$), mais il remplit cette fois la condition de passer une fois dans chaque ville. #

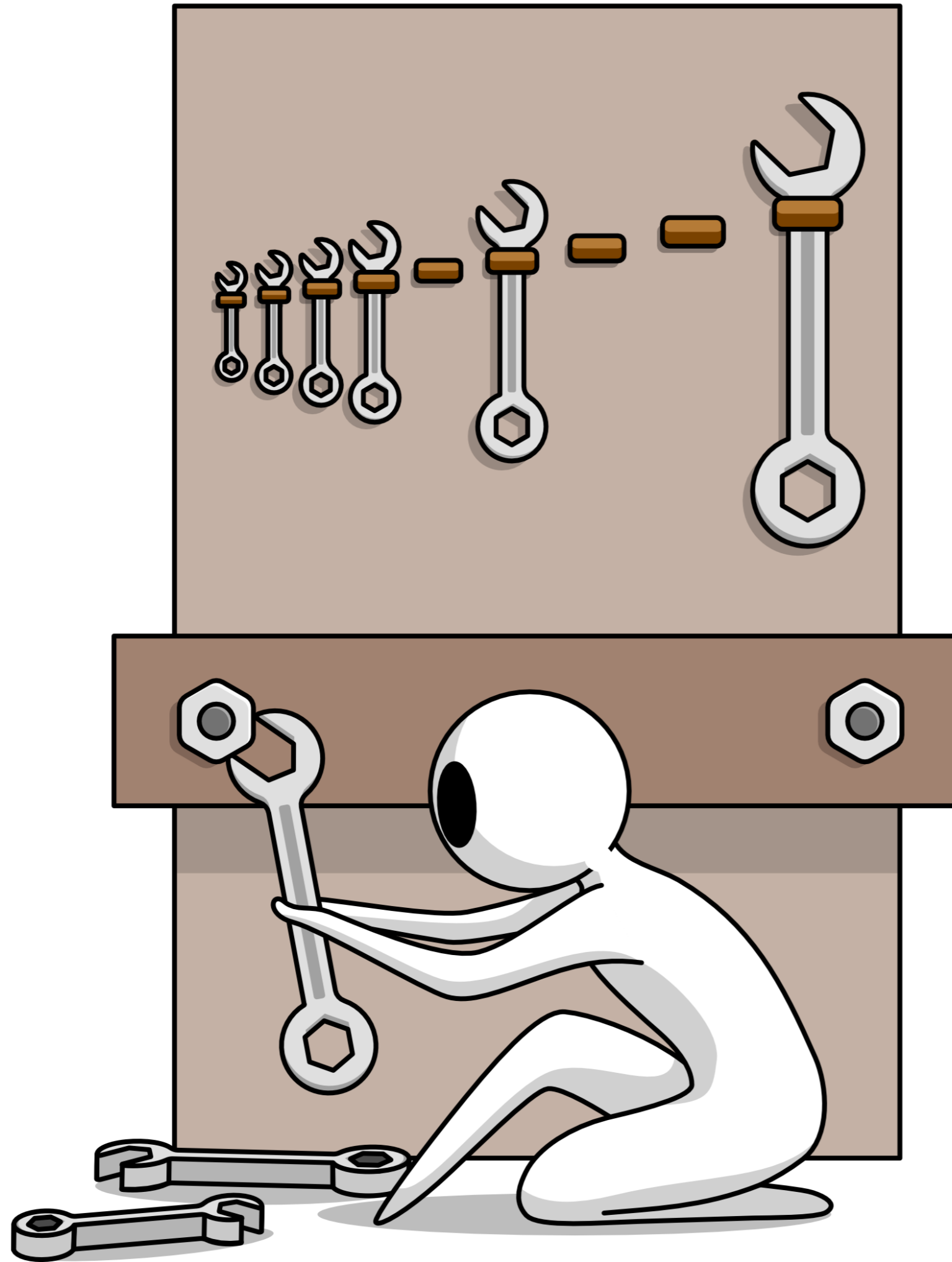
Et en voilà un qui résout le problème tout seul!



Source: “Computation of the Travelling Salesman Problem by a Shrinking Blob”
<https://arxiv.org/abs/1303.4969>

Il s’agit d’un “blob” (plus savamment, un *physarum polycephalum*), organisme unicellulaire capable d’optimiser sa structure interne pour accéder à de la nourriture.

Utilisation du hasard en algorithmique



Algorithmique et programmation avancée

Recherche de la médiane

Recherche de la médiane dans une liste

Soit L une liste de n nombres. On définit:

- d'une part, sa **moyenne**:
$$m = \frac{L(1)+L(2)+ \dots +L(n)}{n}$$
- d'autre part, sa **médiane**: c'est un nombre m_L de la liste L tel que:
 - au moins la moitié des nombres de L sont plus grands ou égaux à m_L
 - au moins la moitié des nombres de L sont plus petits ou égaux à m_L

Exemple: Si $L = (12, 17, 15, 55, 18)$, alors $m = 23,4$ et $m_L = 17$,
car 3 nombres de la liste sont plus ≥ 17 , et 3 nombres sont ≤ 17 .

→ m_L est bien mieux représentative (et fait aussi toujours partie de la liste !)

Recherche de la médiane dans une liste

Cependant:

- Il existe un algorithme simple pour calculer la moyenne.
- Mais quel algorithme utiliser pour calculer efficacement la médiane ?

(à votre tour !)

Une solution pas optimale, mais honnête...

1. Trier la liste L avec votre algorithme de tri favori (par insertion, par fusion)
2. Choisir l'élément numéro $\frac{n}{2}$ de cette liste ($\frac{n+1}{2}$ si n est impair).

Quel est le défaut principal de cet algorithme ?

Il fait *beaucoup plus* que résoudre la question posée !

Une autre solution

Généralisons un peu le problème:

Trouver le nombre de rang k dans la liste L (c'est-à-dire le k^e nombre le plus petit de la liste L ; si $L = (12, 17, 15, 55, 18)$ et $k = 2$, alors la réponse est 15).

- Si k est petit (i.e. égal à 1,2,3, ...) ou au contraire proche de n , il y a de nouveau une solution facile à ce problème: voyez-vous laquelle ?
- Et si maintenant k est plutôt proche de $\frac{n}{2}$, quel est le problème ce cette solution facile ?

Faire appel au hasard...

- Choisissons *uniformément au hasard* un nombre p (comme “pivot”) dans la liste L .
- De là, contruisons trois sous-listes:
 - L_1 , composée des nombres de L plus petits que p
 - L_2 , composée des nombres de L égaux à p
 - L_3 , composée des nombres de L plus grands que p
- Si $k \leq |L_1|$, l’algorithme se relance sur la liste L_1 , avec le même rang k
- Si $|L_1| < k \leq |L_1| + |L_2|$, l’algorithme sort le nombre p
- Si $k > |L_1| + |L_2|$, l’algorithme se relance sur la liste L_3 , avec le rang $k' = k - |L_1| - |L_2|$.

recherche de la médiane

entrée : Liste L non-triée de nombres entiers, de taille n ; rang k

sortie : k^e plus petit nombre de la liste L

$p \leftarrow$ pivot choisi uniformément au hasard dans la liste L

$L_1 \leftarrow$ sous-liste de L composée des nombres plus petits que p

$L_2 \leftarrow$ sous-liste de L composée des nombres égaux à p

$L_3 \leftarrow$ sous-liste de L composée des nombres plus grands que p

Si $k \leq |L_1|$, sortir **recherche de la médiane**(L_1, k)

Si $|L_1| < k \leq |L_1| + |L_2|$, sortir p

Si $k > |L_1| + |L_2|$, sortir **recherche de la médiane**($L_3, k - |L_1| - |L_2|$)

Complexité temporelle moyenne

- A chaque étape, la création des trois sous-listes demande $\Theta(n)$ opérations.
- Et dans le *pire* des cas, le choix du pivot p peut toujours être malchanceux...
...de telle sorte que n étapes sont nécessaires avant que l'algorithme termine.
- En conclusion, une complexité temporelle $\Theta(n^2)$, *dans le pire des cas*.
- Mais en réalité, le choix du pivot p est bon dans au moins 50% des cas (car justement, celui-ci est choisi uniformément au hasard à chaque étape).
- De là, on peut déduire qu'*en moyenne*, la complexité temporelle de l'algorithme vaut $\Theta(n)$... *plus efficace que de trier la liste!*



Algorithmique et programmation avancée

Tri rapide

Olivier Lévêque

Tri rapide

- De la recherche de la médiane, nous pouvons facilement obtenir maintenant un nouvel algorithme de tri de la liste L : le **tri rapide** ("quicksort").
- L'idée consiste à nouveau à choisir un pivot p au hasard dans la liste, puis à séparer la liste L en trois sous-listes L_1, L_2, L_3 comme auparavant, et à récursivement trier les sous-listes L_1 et L_3 avec le même algorithme.

tri rapide

entrée : Liste L non-triée de nombres entiers (de taille n)

sortie : Liste L' triée

Si $n = 1$ ou $n = 0$ (liste vide), sortir : L

$p \leftarrow$ pivot choisi uniformément au hasard dans la liste L

$L_1, L_2, L_3 \leftarrow$ les trois sous-listes générées comme décrit précédemment

$L_1' \leftarrow$ **tri rapide**(L_1)

$L_3' \leftarrow$ **tri rapide**(L_3)

Sortir : $L' = (L_1', L_2, L_3')$

Complexité temporelle moyenne

- On peut montrer que celle-ci vaut $\Theta(n \cdot \log_2(n))$.
- Donc un tri aussi efficace que le tri par fusion.
- Mais notez une différence importante:
 - Dans le tri par fusion, tout le travail de tri est exécuté lors des différentes *fusions* des sous-listes.
 - Dans le tri rapide, tout le travail de tri est effectué lors des différentes *séparations* des sous-listes.