

IN 101 - Cours 05

7 octobre 2011



présenté par

Matthieu Finiasz

- ✘ Le paradoxe des anniversaires dit que $\sqrt{365}$ élèves sont suffisants (en moyenne) pour avoir une **collision d'anniversaire**,
→ deux élèves ayant leur anniversaire le même jour.
- ✘ Comment fait-on pour **efficacement** trouver ces “paires” d’élèves ?

- ✘ Le paradoxe des anniversaires dit que $\sqrt{365}$ élèves sont suffisants (en moyenne) pour avoir une **collision d'anniversaire**,
 - deux élèves ayant leur anniversaire le même jour.
- ✘ Comment fait-on pour **efficacement** trouver ces “paires” d’élèves ?
- ✘ Méthode simple :
 - ✘ on remplit un tableau avec les n dates d’anniversaire,
 - ✘ on compare chaque élément à tous les autres du tableau
 - complexité en $\Theta(n^2)$.
- ✘ Peut-on faire mieux ?

- ✘ Le paradoxe des anniversaires dit que $\sqrt{365}$ élèves sont suffisants (en moyenne) pour avoir une **collision d'anniversaire**,
 - deux élèves ayant leur anniversaire le même jour.
- ✘ Comment fait-on pour **efficacement** trouver ces “paires” d’élèves ?
- ✘ Méthode simple :
 - ✘ on remplit un tableau avec les n dates d’anniversaire,
 - ✘ on compare chaque élément à tous les autres du tableau,
 - complexité en $\Theta(n^2)$.
- ✘ Peut-on faire mieux ? → OUI
 - ✘ on **trie le tableau**,
 - ✘ on le parcourt en regardant si 2 voisins sont égaux,
 - complexité en $\Theta(n \log n)$.

Comment trier un tableau

- ✘ On se donne un tableau de n éléments (des entiers par exemple) et une relation d'ordre totale \leq .
 - ✘ on effectue un tri par comparaison, utilisant uniquement \leq
 - ✘ la complexité est le nombre de comparaisons.

- ✘ Les algorithmes élémentaires ont une complexité en $\Theta(n^2)$.
- ✘ Les meilleurs algorithmes ont une complexité en $\Theta(n \log(n))$.

En autorisant plus que des comparaisons, on peut parfois faire $\Theta(n)$.

Complexité dans le pire cas

Temps de calcul dans le pire cas pour les entrées de taille n fixée :

$$T(n) = \max_{\{x, |x|=n\}} T(x).$$

Complexité moyenne

Temps de calcul moyen sur toutes les entrées de taille n fixée :

$$T_m(n) = \sum_{x, |x|=n} p_n(x) T(x).$$

(p_n) est une distribution de probabilité sur les entrées de taille n .

Si ça marche pour n , ça marche pour $n + 1$

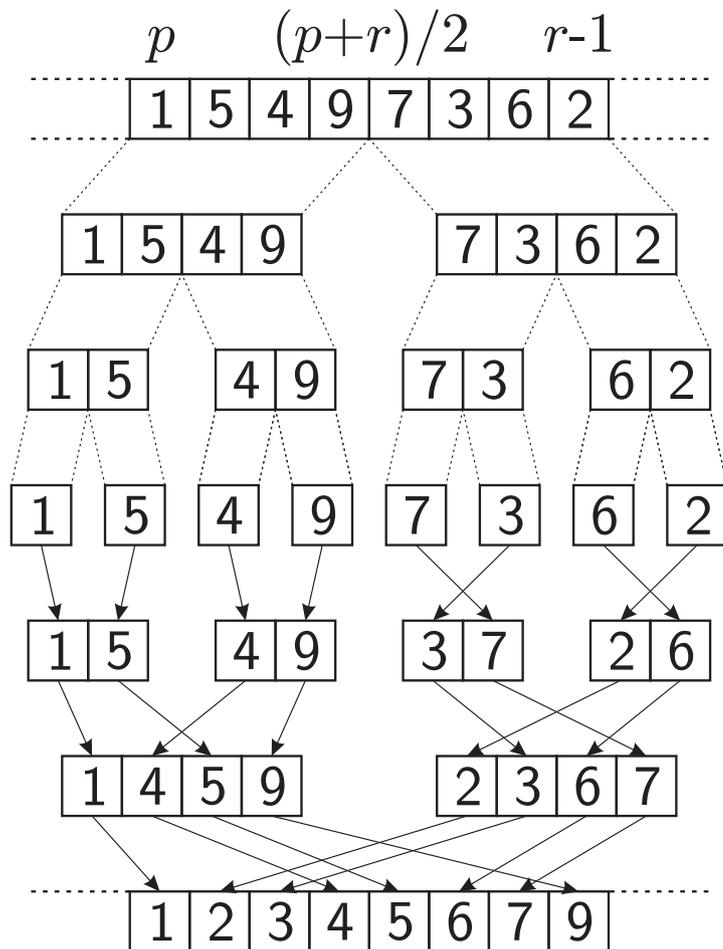
```
1 void insertion_sort(int* tab, int n) {
2     int i,j,tmp;
3     for (i=1; i<n; i++) {
4         tmp = tab[i];
5         j = i-1;
6         while ((j >= 0) && (tab[j] > tmp)) {
7             tab[j+1] = tab[j];
8             j--;
9         }
10        tab[j+1] = tmp;
11    }
12 }
```

- ✘ L'élément i est rangé parmi les $i - 1$ premiers éléments (déjà triés) :
 - ✘ insérer un élément coûte au pire i en moyenne $\frac{i}{2}$,
 - ✘ complexité en $\Theta(n^2)$ dans le pire cas (tableau inversement trié),
 - ✘ complexité moyenne en $\Theta(n^2)$ aussi.

- ✘ L'idée est de faire remonter les grands éléments à la fin du tableau
 - ✘ on parcourt le tableau en comparant l'élément i au $i + 1$ et on les inverse si nécessaire,
 - à la fin du parcours, le dernier élément est le plus grand, après i parcours les i plus grands éléments sont à la fin.
 - ✘ on fait $n - 1$ parcours et la tableau est trié.
- ✘ Complexité : on effectue $\Theta(n)$ parcours comportant $\Theta(n)$ comparaisons à chaque fois,
 - ✘ le tri à bulles a une complexité de $\Theta(n^2)$ dans le pire cas et en moyenne.
- ✘ Pour améliorer un peu les performances, on arrête le tri dès que l'un des parcours n'inverse aucun éléments.
 - ✘ Ce tri ne fait que $\Theta(n)$ comparaisons sur un tableau déjà trié.

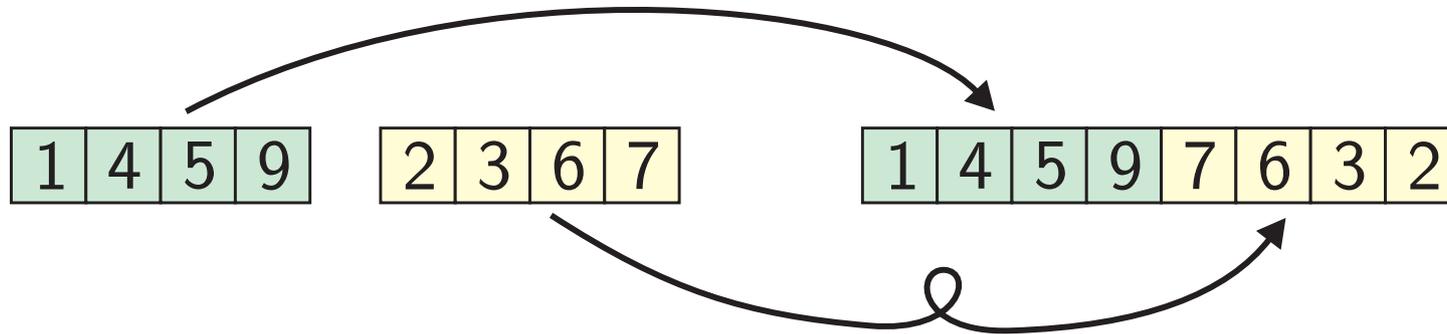
Si ça marche pour $\frac{n}{2}$, ça marche pour n

- ✘ Insérer un éléments dans un tableau trié coûte $\Theta(n)$,
- ✘ fusionner deux tableaux de taille $\frac{n}{2}$ coûte aussi $\Theta(n)$.

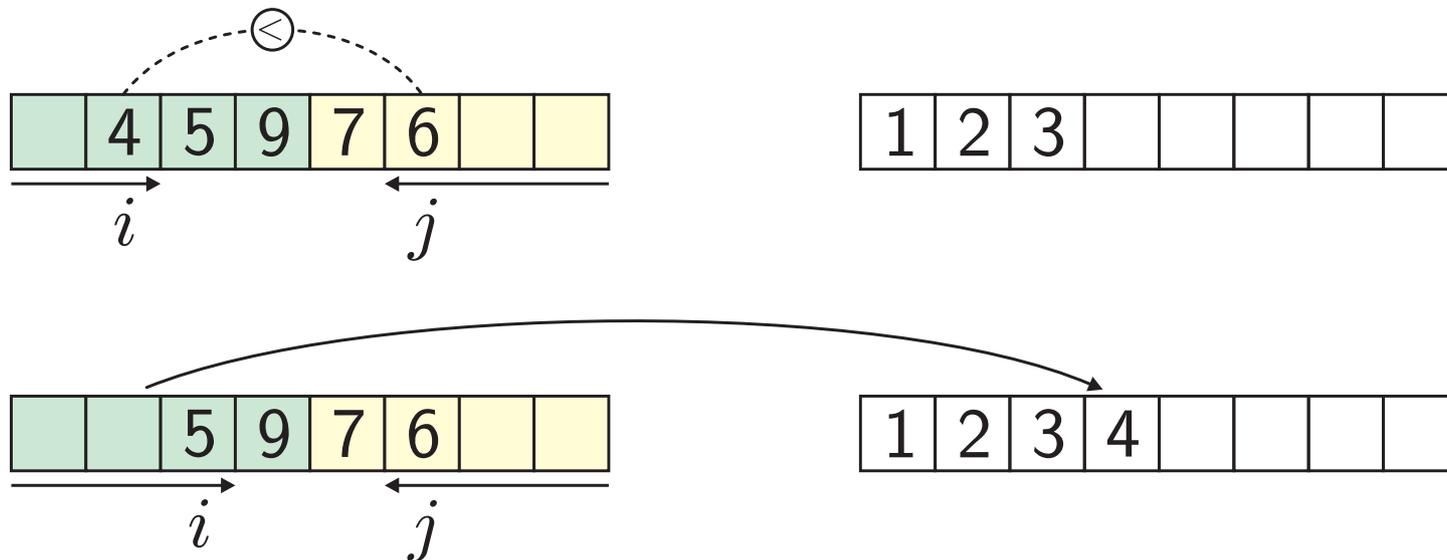


- ✘ On cherche à réduire le problème :
 - ✘ on coupe le tableau en deux,
 - ✘ on trie chaque moitié,
 - ✘ on fusionne.
- ✘ Le coût total est celui des fusions :
 - ✘ chaque fusion coûte $\Theta(r - p)$,
 - ✘ le coût total est $\Theta(n \log(n))$,
 - ✘ dans le pire cas et en moyenne.

- ✘ On commence par recopier “dos à dos” les 2 tableaux (coût $\Theta(n)$),



- ✘ puis on parcourt par les deux bouts en avançant du plus petit côté à chaque fois (coût $\Theta(n)$ aussi).



- ✘ Tri récursif basé sur un partitionnement :
 - ✘ on choisit un **pivot**,
 - ✘ on permute les éléments du tableau
 - les petits au début, puis le pivot, puis les grands,
 - ✘ on trie les petits entre eux et les grands entre eux.

```
1 void quick_sort(int* tab, int p, int r) {
2   if (r-p > 1) {
3     int q = partition(tab,p,r);
4     quick_sort(tab,p,q);
5     quick_sort(tab,q+1,r);
6   }
7 }
```

- ✘ Très rapide, peut être fait **en place** :
 - ✘ complexité dans le pire cas $\Theta(n^2)$,
 - ✘ complexité en moyenne $\Theta(n \log(n))$.

✘ La fonction partition effectue $r - p - 1$ comparaisons.

```
1 int partition(int* tab, int p, int r) {
2     int x = tab[p];
3     int q = p;
4     int i,tmp;
5     for (i=p+1; i<r; i++) {
6         if (tab[i] <= x) {
7             q++;
8             tmp = tab[q];
9             tab[q] = tab[i];
10            tab[i] = tmp;
11        }
12    }
13    tmp = tab[q];
14    tab[q] = tab[p];
15    tab[p] = tmp;
16    return q;
17 }
```

Algorithme	Cas le pire	En moyenne
Tri par insertion Tri à bulles	$\Theta(n^2)$	$\Theta(n^2)$
Tri rapide	$\Theta(n^2)$	$\Theta(n \log(n))$
Tri fusion	$\Theta(n \log(n))$	$\Theta(n \log(n))$

En moyenne

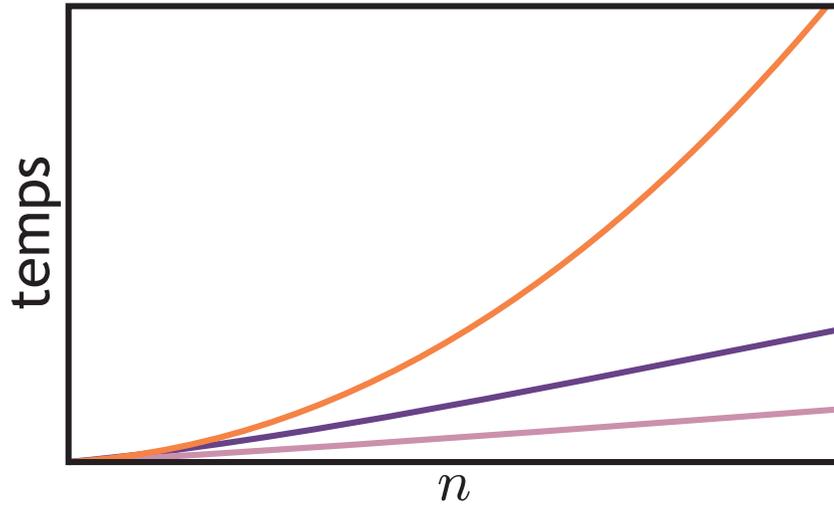


Tableau inversement trié

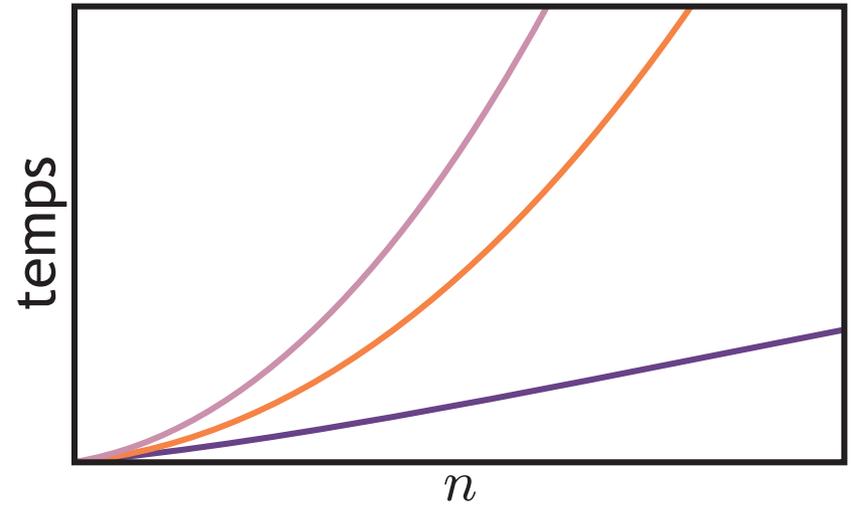
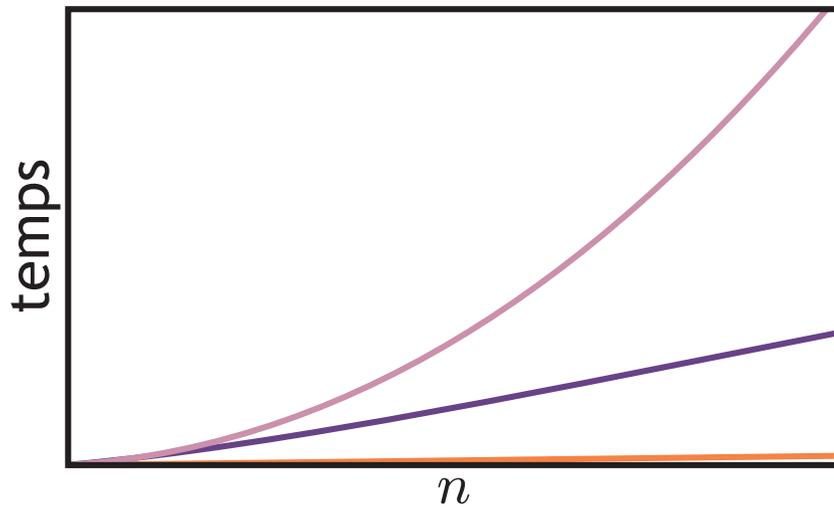


Tableau trié



- Tri rapide
- Tri par insertion
- Tri fusion

- ✘ Pour trouver la médiane (ou le k -ième élément) d'un tableau :
 - ✘ on peut trier le tableau et prendre l'élément $\frac{n}{2}$,
 - coûte $\Theta(n \log n)$.
 - ✘ mais cela n'est pas nécessaire...

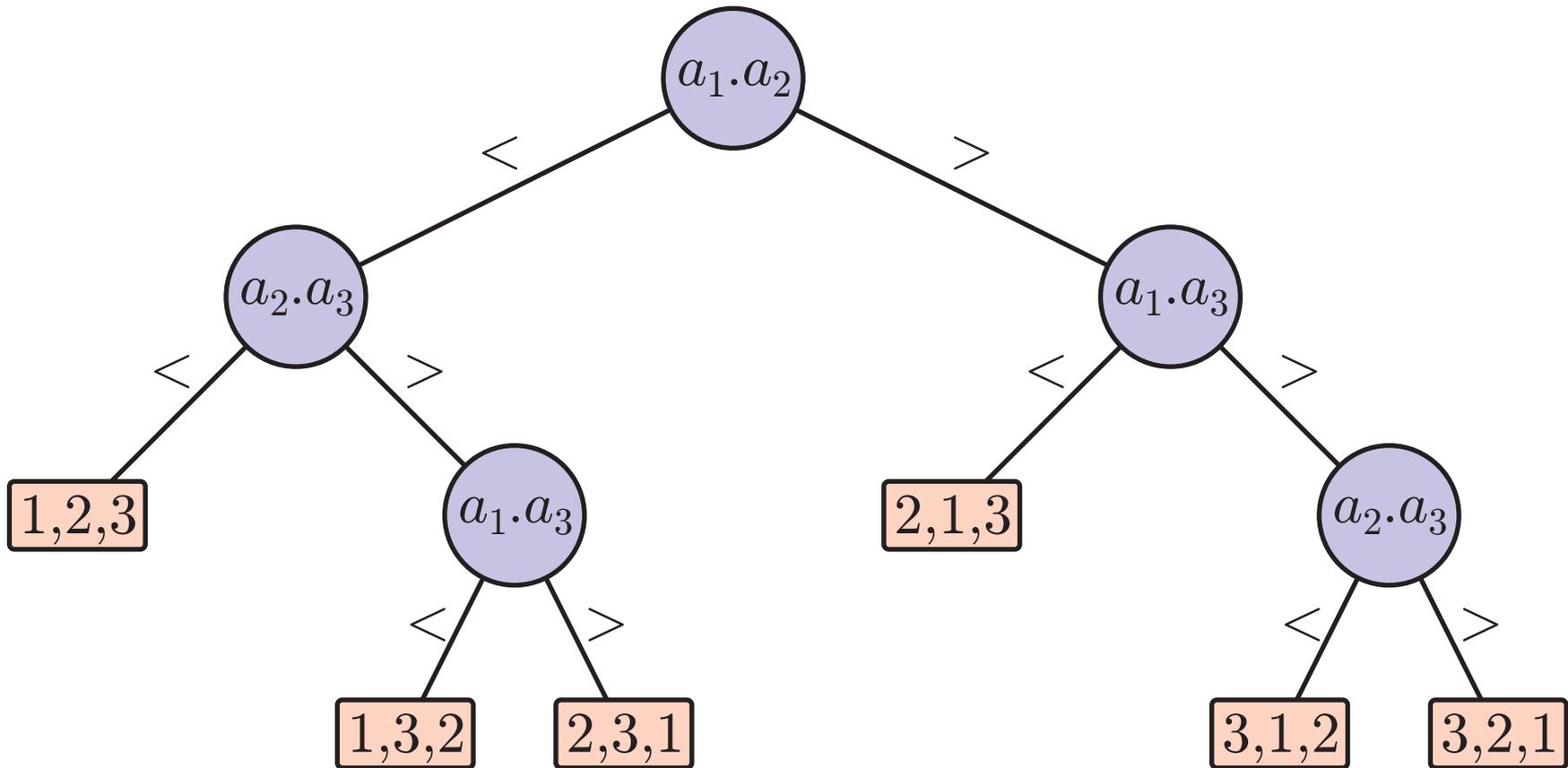
- ✘ On s'inspire du tri rapide :
 - ✘ on conserve la ligne : `int q = partition(tab, p, r);`
 - ✘ au lieu de 2 appels récurifs, on n'en fait qu'un
 - selon que $q < \frac{n}{2}$ ou $q > \frac{n}{2}$,
 - ✘ on ne fait qu'une partie du tri, le minimum nécessaire,
 - la complexité moyenne est $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots = 2n = \Theta(n)$.

Complexité minimale d'un algorithme de tri

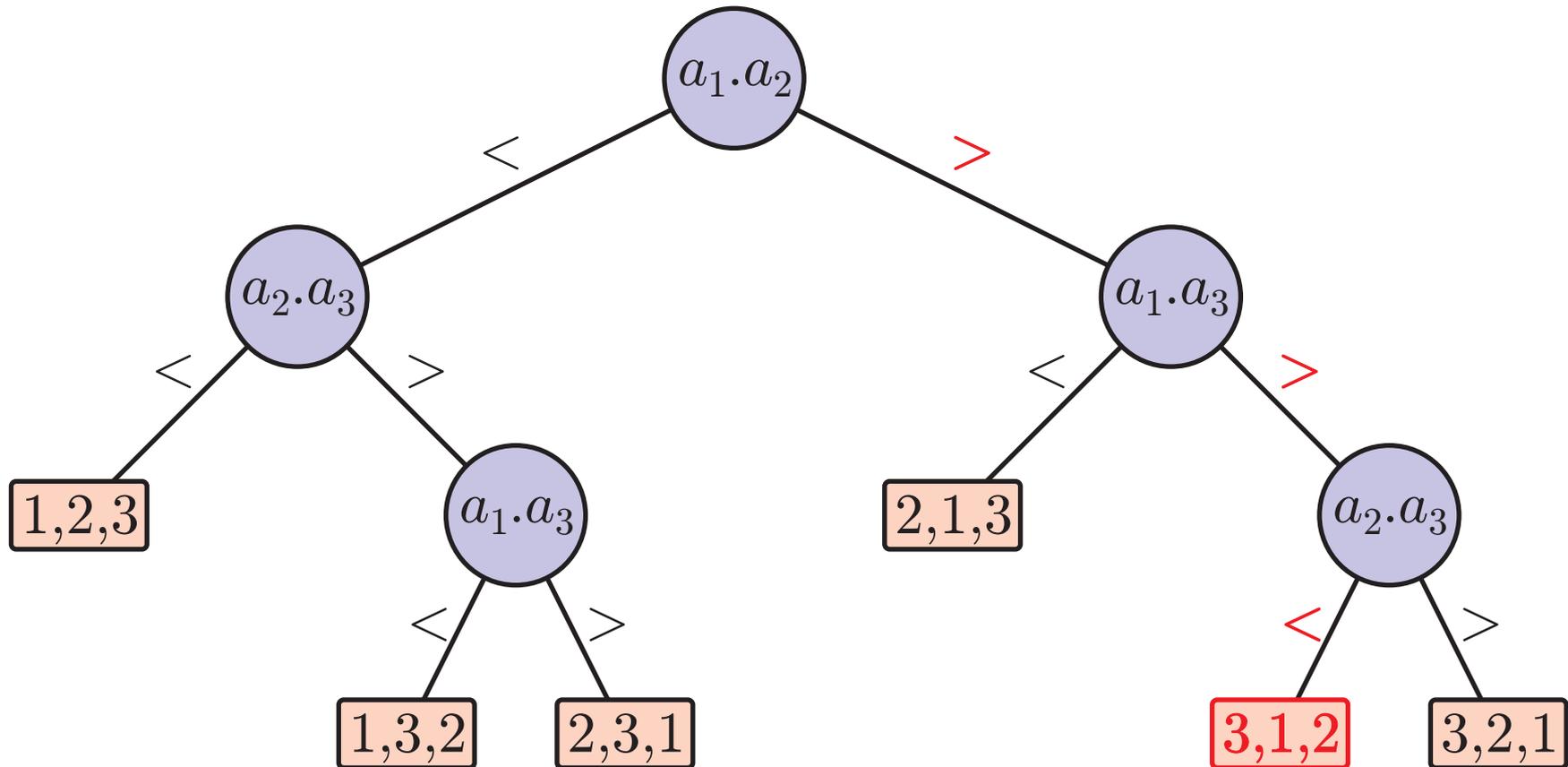
- ✘ Un tri par comparaison prend en entrée n éléments
 - ✘ ne fait que des comparaisons $\rightarrow \{1,4,3\}$ est **identique** à $\{2,4,3\}$
 - ✘ $n!$ entrées possibles, selon l'ordre des éléments.
- ✘ Un algorithme de tri par comparaison fait une suite de comparaison, puis donne une permutation qui remet les éléments en ordre
 - \rightarrow se comporte différemment pour chacune des $n!$ entrées possibles.
- ✘ Un tel tri peut se représenter par un **arbre binaire** :
 - ✘ chaque nœud est une comparaison
 - ✘ en fonction du résultat on descend dans le fils gauche ou droit
 - ✘ chaque feuille correspond à une permutation
 - \rightarrow chaque entrée aboutit à une feuille différente.

Complexité minimal d'un algorithme de tri

Exemple pour $n = 3$



Complexité minimal d'un algorithme de tri

Exemple pour $n = 3$ 

✘ Nombre de comparaison = longueur de la branche

→ cet arbre représente un algorithme qui trie en 3 comparaisons.

Complexité minimale d'un algorithme de tri

$\Theta(n \log n)$, pas mieux

- ✘ Un arbre de hauteur moyenne h contient au plus 2^h feuilles

$$2^h \geq \text{nombre de feuilles} = n!$$

$$h \geq \log(n!) > \log\left(\frac{n^n}{e^n}\right) = n \log n - n \log e$$

- ✘ h est aussi le nombre moyen de comparaisons
 - ✘ pour des entrées équidistribuées
 - la complexité moyenne d'un tri est $\Omega(n \log n)$.
 - ✘ pour n'importe quelles entrées
 - la complexité dans le pire cas est $\Omega(n \log n)$.

Récurtivité

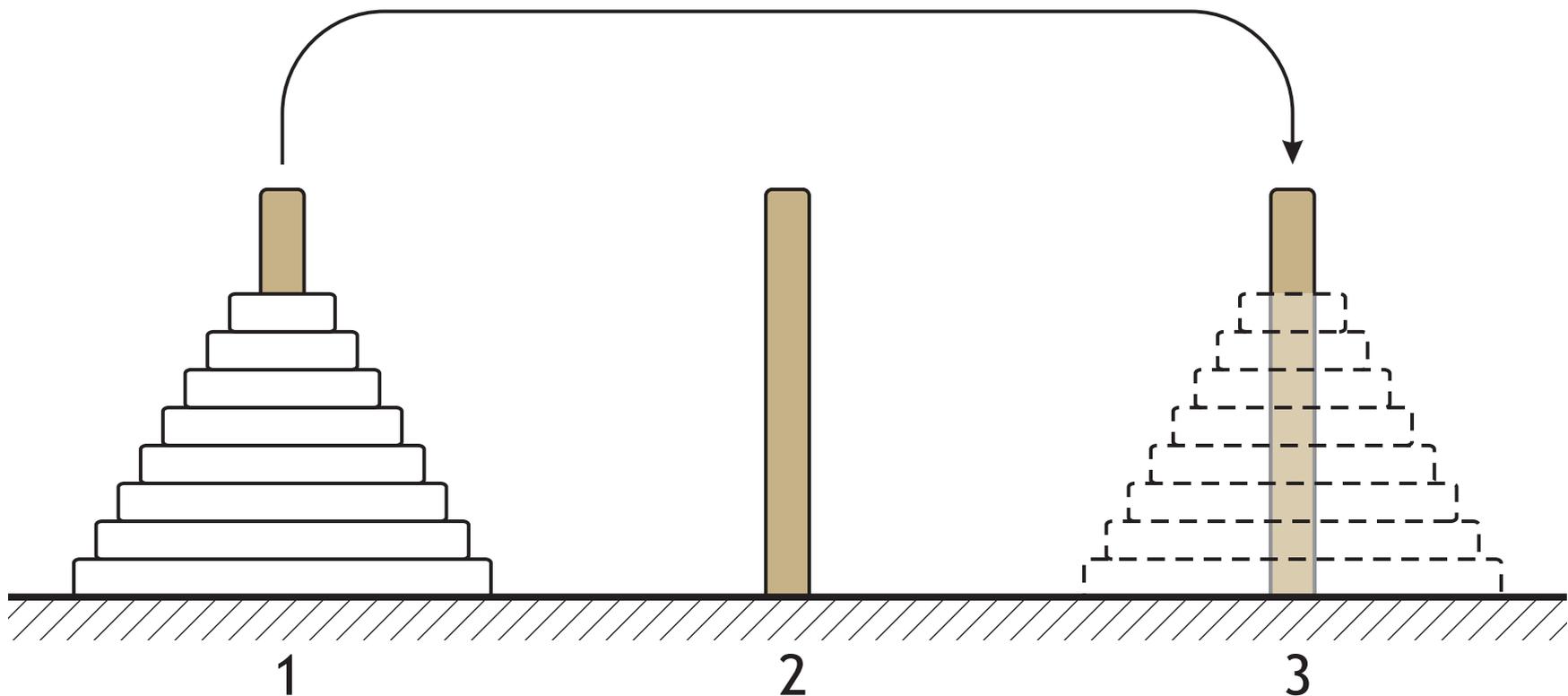
IN101 - 2011-2012

*Un algorithme **récursif** est un algorithme défini en référence à lui même, et qui comprend une **condition de terminaison**.*

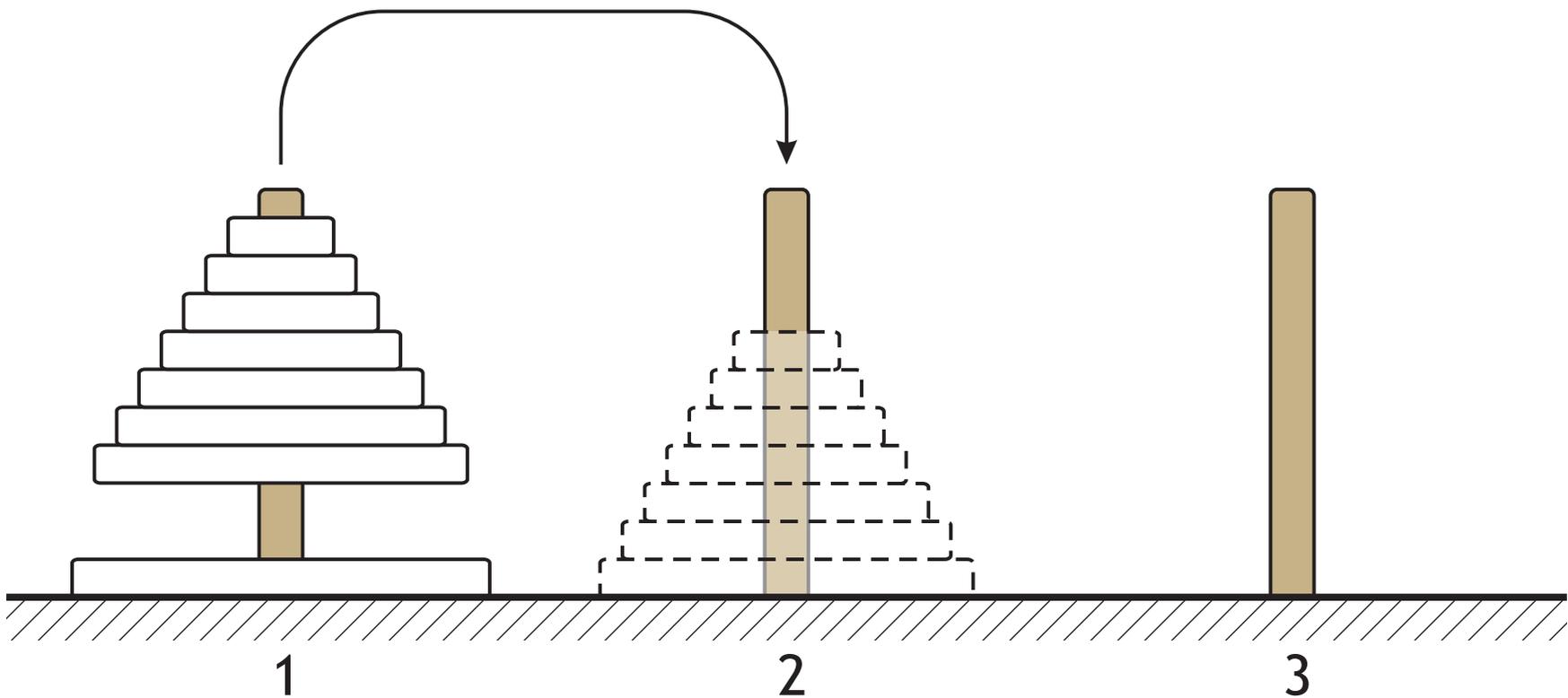
- ✘ Exemples : suite de Fibonacci, tri fusion, tri rapide...
 - ✘ souvent faciles à écrire et à comprendre
 - ✘ c'est le compilateur qui fait le travail
 - ✘ très différent d'une fonction normale !

- ✘ Permettent d'utiliser une approche « **diviser pour régner** » :
 - ✘ on divise le problème en sous-problèmes
 - ✘ on traite les sous-problèmes (récursivement ou directement)
 - ✘ on recombine les résultats.

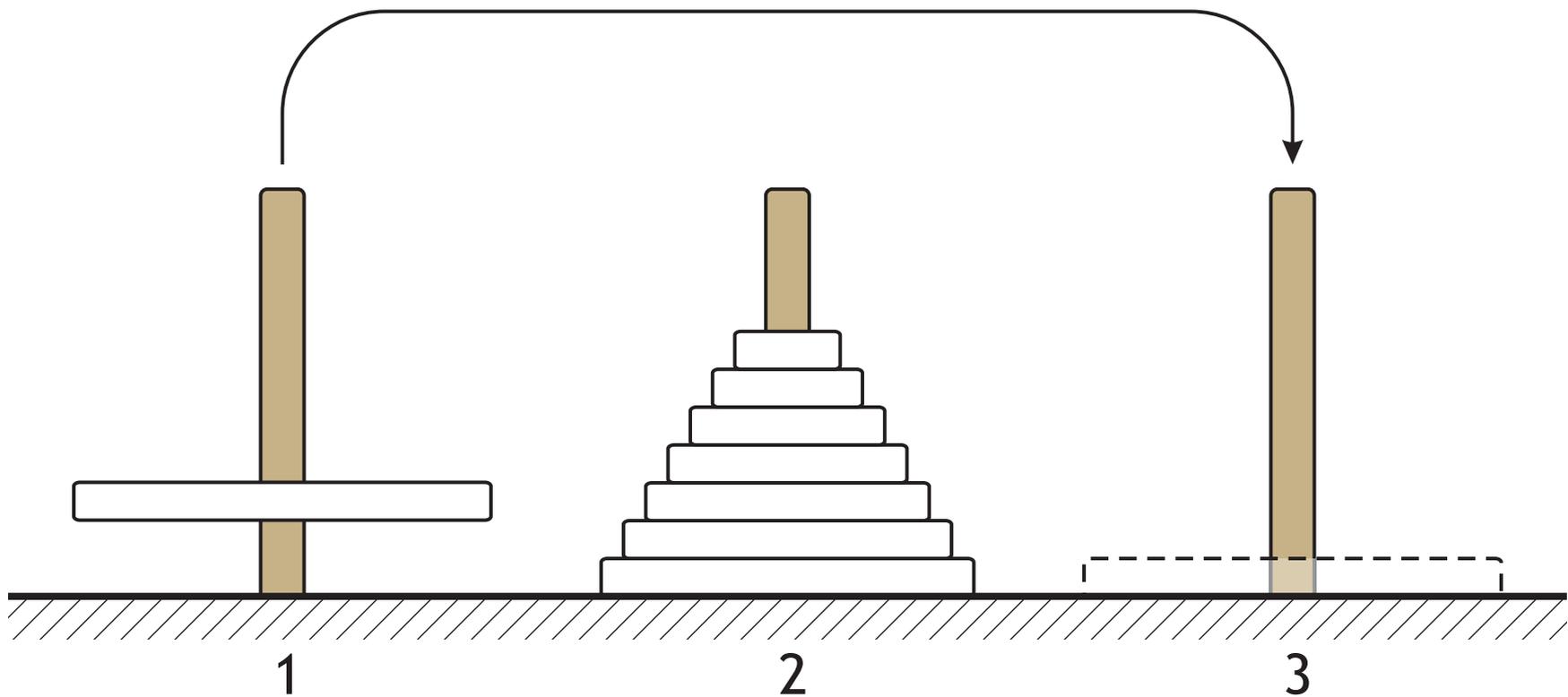
- ✘ On cherche à déplacer n disques de la tour 1 à la tour 3
 - ✘ il est interdit de poser un disque sur un plus petit que lui
 - ✘ on déplace un seul disque à la fois.



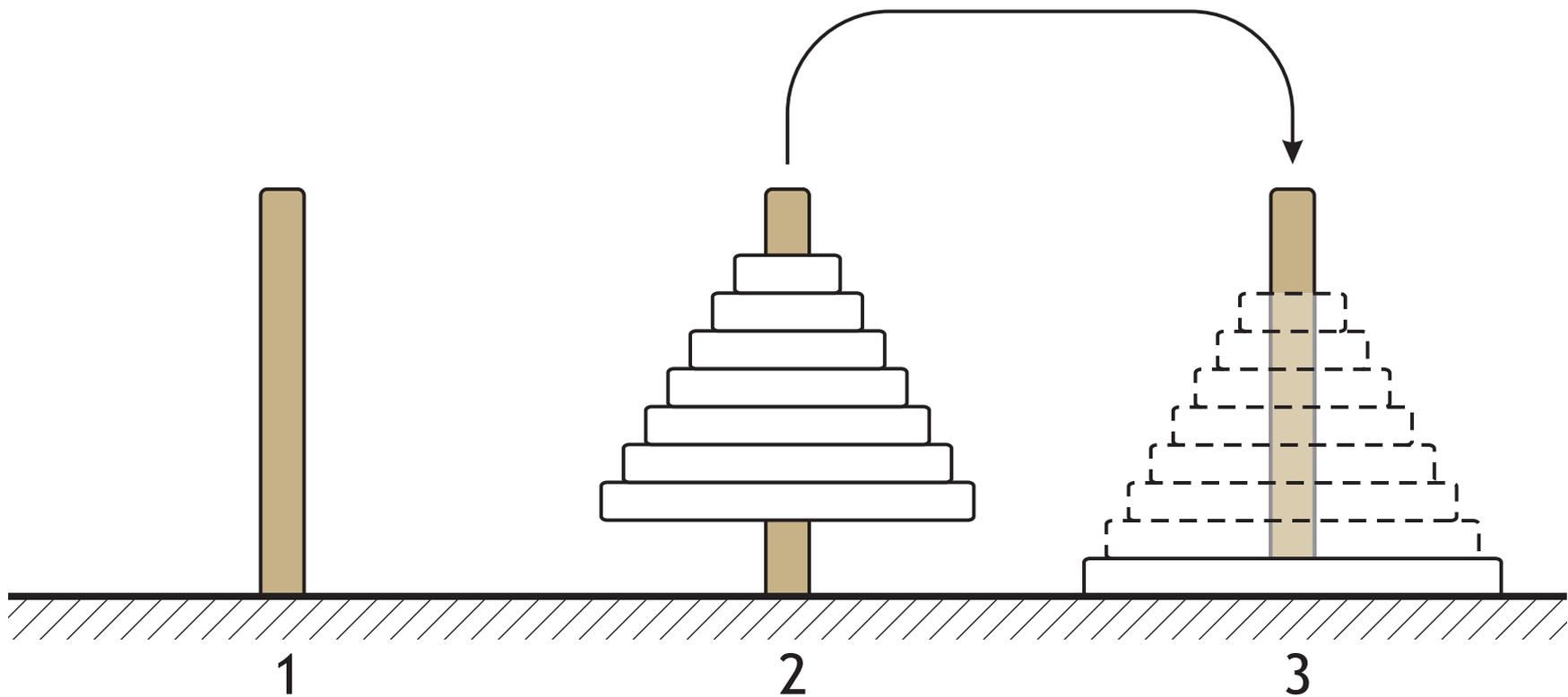
- ✘ On déplace **récurivement** $n - 1$ disques vers la tour 2.



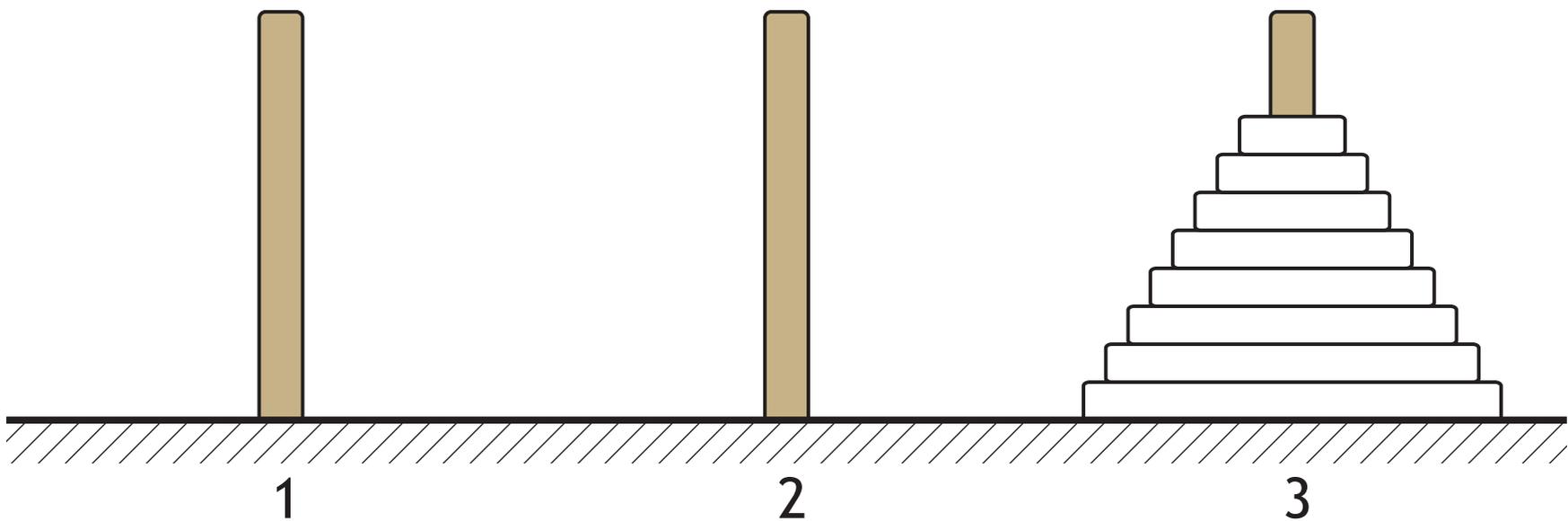
- ✘ Puis on déplace le grand disque à sa place.
- ⚠ Cette étape est **nécessaire** dans tout solution.



- ✘ On redéplace **récurivement** les $n - 1$ petits disques vers la tour 3.



- ✘ Et voilà ! La magie de la récursion a encore une fois opérée...
 - ✘ on décrit en quelques étapes un algorithme exponentiel en n .



- ✘ Cet algorithme affiche les opérations à effectuer :

```
1 void Hanoi(int n, int i, int j) {
2     int intermediate = 6-(i+j);
3     if (n > 0) {
4         Hanoi(n-1,i,intermediate);
5         printf("Mouvement de %d vers %d\n",i,j);
6         Hanoi(n-1,intermediate,j);
7     }
8 }
9
10 int main(int argc, char* argv[]) {
11     Hanoi(atoi(argv[1]),1,3);
12 }
```

- ✘ $T(n)$ le nombre de mouvements nécessaire pour déplacer n disques :
 - ✘ $T(0) = 0$ et $T(1) = 1$,
 - ✘ $T(n) = 1 + 2 \times T(n - 1)$.

→ On obtient : $T(n) = 2^n - 1 = \Theta(2^n)$.

- ✘ L'étape de mouvement du grand disque est nécessaire
 - ✘ cette complexité est **intrinsèque** au problème.
- ✘ La complexité spatiale est $\Theta(n)$
 - ✘ on ne stocke rien, mais on a n appels récursifs imbriqués.

Pour déplacer 50 disques, à 1 disque par seconde, il faut environ 35 millions d'années.

Complexité d'algorithmes récur­sifs

- ✘ Calculer la complexité $T(n)$ en nombre de comparaisons :

$$T(n) = D(n) + R(n) + C(n).$$

- ✘ $D(n)$: coût pour **diviser le problème** en sous-problèmes. $D(n) = 0$
- ✘ $R(n)$: coût pour **résoudre les sous-problèmes**. Ici, deux sous-problèmes de taille $\frac{n}{2}$, donc $R(n) = 2 \times T(\frac{n}{2})$.
- ✘ $C(n)$: coût pour **combinaison des résultats**. Ici, une fusion : $C(n) = n$.

$$\forall n \geq 2, T(n) = 2 \times T\left(\frac{n}{2}\right) + n$$

- ✘ Une telle équation admet comme solution $T(n) = \Theta(n \log n)$.

```
1 void quick_sort(int* tab, int p, int r) {
2     int q;
3     if (r-p > 1) {
4         q = partition(tab,p,r);
5         quick_sort(tab,p,q);
6         quick_sort(tab,q+1,r);
7     }
8 }
```

- ✘ `partition(tab, p, r)` nécessite $r - p - 1$ comparaisons,
→ on calcule la complexité de `quick_sort` comme précédemment.

✘ Pour une entrée de taille n :

- ✘ Division en deux sous-problèmes de taille $q - 1$ et $n - q$, coût de partition : $D(n) = n - 1$.
- ✘ Résolution des sous-problèmes : coût $R(n)$ variable selon la valeur de q .
- ✘ Combinaison des résultats : coût $C(n) = 0$.

$$T(n) = n - 1 + \begin{cases} 2 \times T\left(\frac{n}{2}\right) & \text{meilleur cas} \\ \frac{1}{n} \sum_{q=1}^n T(q - 1) + T(n - q) & \text{cas moyen} \\ T(n - 1) & \text{pire cas} \end{cases}$$

✘ On trouve les solutions :

$$T(n) = \Theta(n \log n) \text{ (meilleur cas et cas moyen),}$$

$$T(n) = \Theta(n^2) \text{ (pire cas).}$$

```
1 unsigned int fibo1(unsigned int n) {
2     if (n < 2) {
3         return n;
4     }
5     return fibo1(n-1) + fibo1(n-2);
6 }
```

✘ Si $T(n)$ est la complexité du calcul du n -ième nombre :

$$T(n) = \underbrace{T(n-1) + T(n-2)}_{R(n)} + \underbrace{1}_{C(n)}$$

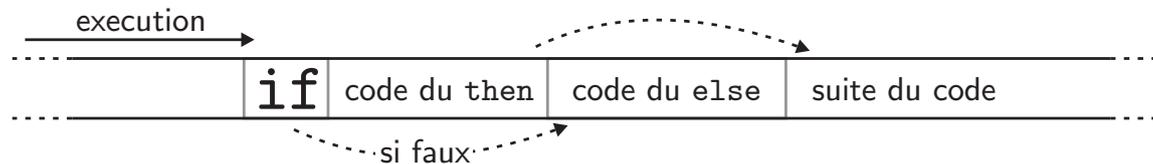
✘ on fait un **changement de variable** $S(n) = T(n) + 1$ qui donne
 $S(n) = S(n-1) + S(n-2)$ et donc $S(n) = \Theta(\varphi^n)$.

✘ Comme annoncé précédemment $T(n) = \Theta(\varphi^n)$.

Les dessous de la récursivité

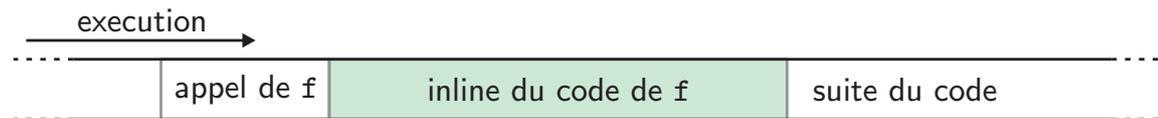
Le code produit par un compilateur est une suite d'instruction avec des sauts, c'est tout.

- ✘ Un `if` est un saut conditionnel :
 - ✘ soit on exécute l'instruction suivante, soit on fait un saut un peu plus loin.

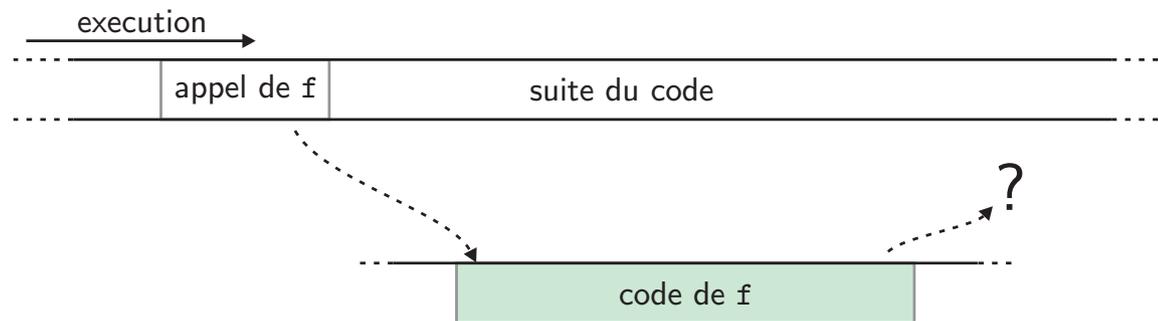


Le code produit par un compilateur est une suite d'instruction avec des sauts, c'est tout.

- ✘ Un appel à une fonction peut-être traité de deux façons :
 - ✘ soit on copie le code de la fonction (c'est de l'**inline**),

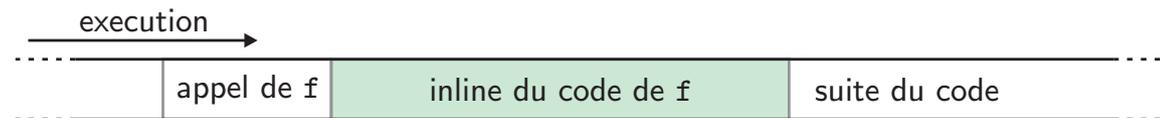


- ✘ soit on fait un saut vers le code de la fonction
→ comment savoir où revenir ?

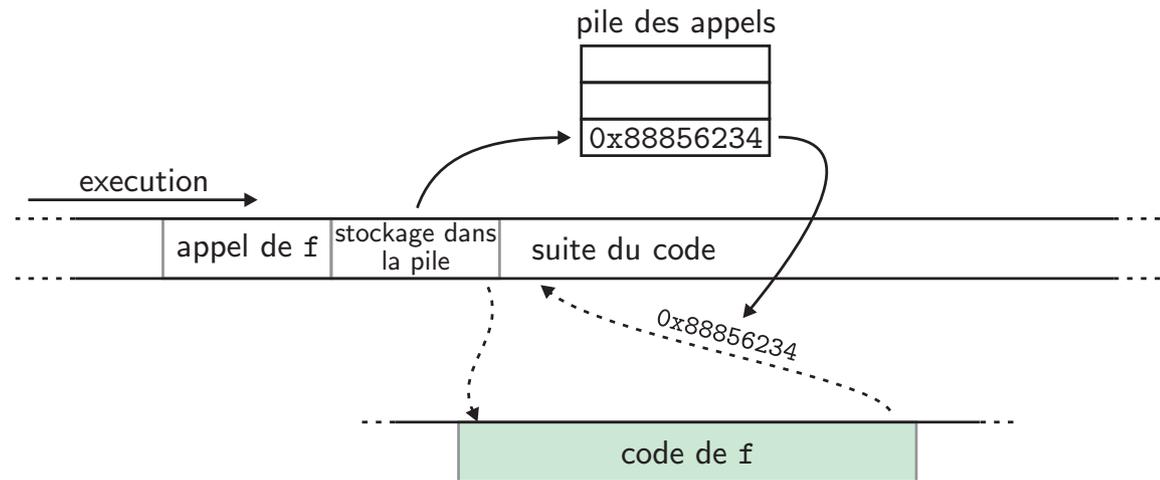


Le code produit par un compilateur est une suite d'instruction avec des sauts, c'est tout.

- ✘ Un appel à une fonction peut-être traité de deux façons :
 - ✘ soit on copie le code de la fonction (c'est de l'**inline**),



- ✘ soit on fait un saut vers le code de la fonction
 - on utilise une pile d'adresses de retour.



Le code produit par un compilateur est une suite d'instruction avec des sauts, c'est tout.

- ✘ Une fonction réursive ajoute successivement les adresses de retour dans la pile :
 - ✘ n appels imbriqués ont une complexité mémoire en $\Theta(n)$.

- ✘ Autres remarques :
 - ✘ un appel à une fonction coûte cher
 - compromis entre inline/taille du code et temps d'exécution,
 - ✘ certains vieux langages ne gèrent pas les fonctions réursives : si on fait toujours de l'inline, les fonctions non réursives sont beaucoup plus simples à compiler que les réursives.

- ✘ La **recursion terminale** est un cas particulier de récursion :
 - ✘ l'appel récursif est la **dernière** commande exécutée,
 - ✘ la valeur de retour de la fonction est la même que celle retournée par l'appel récursif,
 - marche aussi s'il n'y a pas de valeur de retour.

- ✘ Il est alors possible de convertir la fonction récursive en boucle :
 - ✘ un appel de fonction est toujours lourd (écriture dans la pile...)
 - la récursion terminale permet d'être plus efficace.

- ✘ Avec gcc, la récursion terminale est prise en compte quand on utilise l'option d'optimisation -O2 (ou -O3).

```
1 int euclide(int x, int y) {
2     if (y == 0) {
3         return x;
4     } else {
5         return euclide(y, x % y);
6     }
7 }
8 int euclide_iter(int x, int y) {
9     int tmp1, tmp2;
10    while (!(y == 0)) {
11        tmp1 = y;
12        tmp2 = x % y;
13        x = tmp1;
14        y = tmp2;
15    }
16    return x;
17 }
```

- ✘ Il existe beaucoup d'algorithmes différents pour trier un tableau :
 - ✘ les plus basiques coûtent $\Theta(n^2)$,
 - tri par insertion, tri à bulles, tri cocktail...
 - ✘ les meilleurs coûtent $\Theta(n \log n)$,
 - tri fusion, tri rapide, tri par tas, tri par arbre...

- ✘ La récursivité permet de décrire facilement certains algorithmes :
 - ✘ la complexité d'un algorithme récursif est plus difficile à calculer,
 - formules pour les algorithmes "diviser pour régner" (cf. poly)
 - ✘ la compilation d'une fonction récursive est plus compliquée
 - programme un peu plus lent dans certains cas.