Algorithmes Récursifs

Force brute

Force brute

- L'approche "force brute" est une méthode de conception d'algorithmes qui se base simplement sur une énumération exhaustive de toutes les configurations possibles de la solution recherchée

- **Exemple** : Un algorithme de tri de type "force brute " parcourt toutes les permutations possibles de la liste jusqu'à ce qu'il trouve la permutation ordonnée.

Force brute

- L'approche "force brute" est une méthode de conception d'algorithmes qui se base simplement sur une énumération exhaustive de toutes les configurations possibles de la solution recherchée

- **Exemple** : Un algorithme de tri de type "force brute " parcourt toutes les permutations possibles de la liste jusqu'à ce qu'il trouve la permutation ordonnée.

- Cette méthode est souvent inefficace car elle se repose sur l'énumération complète d'un espace de recherche

- **Exemple** (algorithme de tri de type "force brute") : le nombre de permutations possible est $n!$ donc la complexité d'un tel algorithme est $\Theta(n!)$

Diviser pour régner (Divide and conquer)

Diviser pour régner (Divide and conquer)

- “Diviser pour régner” est une méthode de conception d’algorithmes qui se base sur une “conception par décomposition” :

Diviser pour régner (Divide and conquer)

- “Diviser pour régner” est une méthode de conception d’algorithmes qui se base sur une “conception par décomposition” :
  - Diviser le problème en sous problèmes

Diviser pour régner (Divide and conquer)

- “Diviser pour régner” est une méthode de conception d’algorithmes qui se base sur une “conception par décomposition” :
  - Diviser le problème en sous problèmes
  - Résoudre les sous problèmes par des appels *récursifs*

Diviser pour régner (Divide and conquer)

- "Diviser pour régner" est une méthode de conception d'algorithmes qui se base sur une "conception par décomposition" :
  - Diviser le problème en sous problèmes
  - Résoudre les sous problèmes par des appels *récursifs*
  - Combiner les résultats des sous problèmes pour résoudre le problème initial

Diviser pour régner (Divide and conquer)

- "Diviser pour régner" est une méthode de conception d'algorithmes qui se base sur une "conception par décomposition" :
  - Diviser le problème en sous problèmes
  - Résoudre les sous problèmes par des appels *récursifs*
  - Combiner les résultats des sous problèmes pour résoudre le problème initial
- Cas idéal : le problème est décomposable en sous-problèmes *indépendents*
  - Dans ce cas, combiner les résultats est trivial
  - Parfois, les sous-problèmes sont seulement plus "faiblement" liés, et combiner les résultats peut-être complexe

Exemple avec un algorithme de tri

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

Exemple avec un algorithme de tri

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

Algorithme : `Fusion` (L_1, L_2)

Données : deux listes triées L_1 et L_2
Résultat : une liste L triée contenant les éléments de L_1 et de L_2
L : liste vide;
$i, j, k \leftarrow 1$;
tant que $k < |L|$ **faire**
    si $i > |L_1|$ *ou* $(j \leq |L_2|$ *et* $L_1[i] > L_2[j])$ **alors**
        insérer $L_2[j]$ à la fin de L;
        $j \leftarrow j + 1$;
    sinon
        insérer $L_1[i]$ à la fin de L;
        $i \leftarrow i + 1$;
    $k \leftarrow k + 1$;
retourner L

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$
  - Division : $\langle 5 \rangle$ $\langle 4 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$
  - Division : $\langle 5 \rangle$ $\langle 4 \rangle$
  - Regroupement : $\langle 4, 5 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$
  - Division : $\langle 5 \rangle$ $\langle 4 \rangle$
  - Regroupement : $\langle 4, 5 \rangle$
  - Division : $\langle 3 \rangle \langle 7 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$
  - Division : $\langle 5 \rangle$ $\langle 4 \rangle$
  - Regroupement : $\langle 4, 5 \rangle$
  - Division : $\langle 3 \rangle \langle 7 \rangle$
  - Regroupement : $\langle 3, 7 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$
  - Division : $\langle 5 \rangle$ $\langle 4 \rangle$
  - Regroupement : $\langle 4, 5 \rangle$
  - Division : $\langle 3 \rangle \langle 7 \rangle$
  - Regroupement : $\langle 3, 7 \rangle$
  - Regroupement : $\langle 3, 4, 5, 7 \rangle$

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$
  - Division : $\langle 5 \rangle$ $\langle 4 \rangle$
  - Regroupement : $\langle 4, 5 \rangle$
  - Division : $\langle 3 \rangle \langle 7 \rangle$
  - Regroupement : $\langle 3, 7 \rangle$
  - Regroupement : $\langle 3, 4, 5, 7 \rangle$
  - Regroupement : $\langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$

Preuve de correction

- Terminaison :
  - `TriFusion` ne s'appelle lui même que 2 fois
  - A chaque appel récursif, la taille de la liste $|L|$ est strictement plus petite
  - Il n'y a pas d'appel récursif pour $|L| \leq 1 \implies$ **nombre total d'appels récursifs est fini**
- Correction (`TriFusion`(L) est triée, par récurrence sur $|L|$) :
  - Pour $|L| \leq 1$, la liste est déjà triée
  - `TriFusion`(L) triée si $|L| \leq n$; est-ce que `TriFusion`(L) est triée si $|L| \leq n+1$?
  - `TriFusion`(L) renvoie `Fusion`(`TriFusion`$(L[: mil])$, `TriFusion`$(L[mil :])$)
  - `TriFusion`$(L[: mil])$ et `TriFusion`$(L[mil :])$ sont triées par l'hypothèse de récurrence puisque $|L[: mil]| \leq n$ et $|L[mil :]| \leq n$
  - $\Rightarrow$ les préconditions de `Fusion` sont respectées, montront qu'il est correct, par invariants :
    - ★ L est triée
    - ★ $i = |L_1| + 1$ ou $k = 1$ ou $L[k-1] \leq L_1[i]$
    - ★ $j = |L_2| + 1$ ou $k = 1$ ou $L[k-1] \leq L_2[i]$

Analyse de la complexité d'un algorithme récursif

La structure d'un algorithme récursif `AlgoRec(`x`)` est :

si *condition d'arrêt* **alors**
  retourner *solution triviale*;
sinon
  retourner `Fusion(AlgoRec(`$p(x,1)$`),AlgoRec(`$p(x,2)$`),...,AlgoRec(`$p(x,a)$`))`;

Analyse de la complexité d'un algorithme récursif

La structure d'un algorithme récursif `AlgoRec(`x`)` est :

si *condition d'arrêt* **alors**
  retourner *solution triviale*;
sinon
  retourner `Fusion(AlgoRec(`$p(x,1)$`),AlgoRec(`$p(x,2)$`),...,AlgoRec(`$p(x,a)$`))`;

- p "coupe" la donnée x (de taille $|x| = n$) en a morceaux de taille $g(n)$
- `Fusion` "recolle" les morceaux en $h(n)$

Analyse de la complexité d'un algorithme récursif

La structure d'un algorithme récursif `AlgoRec(`x`)` est :

si *condition d'arrêt* **alors**
  retourner *solution triviale*;
sinon
  retourner `Fusion(AlgoRec(`$p(x,1)$`),AlgoRec(`$p(x,2)$`),...,AlgoRec(`$p(x,a)$`))`;

- p "coupe" la donnée x (de taille $|x| = n$) en a morceaux de taille $g(n)$
- `Fusion` "recolle" les morceaux en $h(n)$

Forme récursive de la complexité $T(n) = \begin{cases} \Theta(1) & \text{si } \dots \\ aT(g(n)) + h(n) & \text{sinon} \end{cases}$

Analyse de la complexité d'un algorithme récursif

La structure d'un algorithme récursif `AlgoRec(`x`)` est :

si *condition d'arrêt* **alors**
  retourner *solution triviale*;
sinon
  retourner `Fusion(AlgoRec(`$p(x,1)$`),AlgoRec(`$p(x,2)$`),...,AlgoRec(`$p(x,a)$`))`;

- p "coupe" la donnée x (de taille $|x| = n$) en a morceaux de taille $g(n)$
- `Fusion` "recolle" les morceaux en $h(n)$

Forme récursive de la complexité $T(n) = \begin{cases} \Theta(1) & \text{si } \dots \\ aT(g(n)) + h(n) & \text{sinon} \end{cases}$

On veut trouver une *formule* de forme close

$$T(n) \in \mathcal{O}(f(n))$$

Exemple : `TriFusion`

- `TriFusion` "coupe" la donnée L (de taille $|L| = n$) en 2 morceaux de taille $\frac{n}{2}$
- `Fusion` "recolle" les morceaux en $\Theta(n)$

Exemple : `TriFusion`

- `TriFusion` "coupe" la donnée L (de taille $|L| = n$) en 2 morceaux de taille $\frac{n}{2}$
- `Fusion` "recolle" les morceaux en $\Theta(n)$

Forme récursive de la complexité :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n & \text{sinon} \end{cases}$$

Exemple : `TriFusion`

- `TriFusion` "coupe" la donnée L (de taille $|L| = n$) en 2 morceaux de taille $\frac{n}{2}$
- `Fusion` "recolle" les morceaux en $\Theta(n)$

Forme récursive de la complexité :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T(\frac{n}{2}) + n & \text{sinon} \end{cases}$$

Forme fermée de la complexité :

$$\exists c, n_0 \forall n > n_0 \quad T(n) \leq cn \log n$$
$$\text{càd} \quad T(n) \in \mathcal{O}(n \log n)$$

Méthode par substitution

- Il faut avoir une intuition sur la forme de la solution (`TriFusion` : $\mathcal{O}(n \log n)$)

Méthode par substitution

- Il faut avoir une intuition sur la forme de la solution (`TriFusion` : $\mathcal{O}(n \log n)$)
- On veut montrer que $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \in \mathcal{O}(n \log n)$

Méthode par substitution

- Il faut avoir une intuition sur la forme de la solution (`TriFusion` : $\mathcal{O}(n \log n)$)
- On veut montrer que $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \in \mathcal{O}(n \log n)$
- On montre par récurrence qu'il existe $f(n)$ t.q. $\forall n \; T(n) \leq f(n)$
  - On va en déduire **a posteriori** que $T(n) \in \mathcal{O}(f(n))$

Méthode par substitution

- Il faut avoir une intuition sur la forme de la solution (`TriFusion` : $\mathcal{O}(n \log n)$)
- On veut montrer que $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \in \mathcal{O}(n \log n)$
- On montre par récurrence qu'il existe $f(n)$ t.q. $\forall n \; T(n) \leq f(n)$
  - On va en déduire **a posteriori** que $T(n) \in \mathcal{O}(f(n))$

Attention !

- L'hypothèse de récurrence est $T(n) \leq f(n)$, et **non** $T(n) \in \mathcal{O}(f(n))$
- L'hypothèse de récurrence "pour tout $n \leq k$, $T(n) \in \mathcal{O}(f(n))$" ne veut pas dire grand chose puisque la notation $\mathcal{O}$ est définie pour n arbitrairement grand : **on remplace tous les termes en $\mathcal{O}, \Omega, \Theta$ par une fonction élément de l'ensemble**

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- Il faut montrer que la formule est vraie pour les conditions limites de la récurrence
  pour des données de petite taille, i.e. $n = 1$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- Il faut montrer que la formule est vraie pour les conditions limites de la récurrence pour des données de petite taille, i.e. $n = 1$
- **Problème :** c'est faux pour $n = 1$ car $c \times 1 \times \log 1 = 0 < T(1) = 1$;

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- Il faut montrer que la formule est vraie pour les conditions limites de la récurrence
  pour des données de petite taille, i.e. $n = 1$
- **Problème :** c'est faux pour $n = 1$ car $c \times 1 \times \log 1 = 0 < T(1) = 1$;
- Mais on cherche à montrer la complexité pour des données de grande taille : $n \geq n_0$ et on a le choix pour $n_0 \implies$ vérifier pour $T(2)$ (et $T(3)$)

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- Il faut montrer que la formule est vraie pour les conditions limites de la récurrence
  pour des données de petite taille, i.e. $n = 1$
- **Problème :** c'est faux pour $n = 1$ car $c \times 1 \times \log 1 = 0 < T(1) = 1$;
- Mais on cherche à montrer la complexité pour des données de grande taille : $n \geq n_0$ et on a le choix pour $n_0 \implies$ vérifier pour $T(2)$ (et $T(3)$)
- On peut aussi borner par $f(n) = cn \log n + b$ puisque $cn \log n + b \in \mathcal{O}(n \log n)$
  - Ou même $f(n) = cn \log n + an + b$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

$$T(2) = 2T(1) + 2$$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

$$T(2) = 2T(1) + 2$$

$$T(2) = 2 * 1 + 2 = 4 \leq 2c \log 2 = 2c$$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

$$T(2) = 2T(1) + 2$$

$$T(2) = 2 * 1 + 2 = 4 \leq 2c \log 2 = 2c$$

$$T(2) = 4 \leq 2c$$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

$$T(2) = 2T(1) + 2$$

$$T(2) = 2 * 1 + 2 = 4 \leq 2c \log 2 = 2c$$

$$T(2) = 4 \leq 2c$$

$$c \geq 2$$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

$$T(2) = 2T(1) + 2$$

$$T(2) = 2 * 1 + 2 = 4 \leq 2c \log 2 = 2c$$

$$T(2) = 4 \leq 2c$$

$$c \geq 2$$

- On fait la même chose pour $T(3)$...

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

$$T(2) = 2T(1) + 2$$

$$T(2) = 2 * 1 + 2 = 4 \leq 2c \log 2 = 2c$$

$$T(2) = 4 \leq 2c$$

$$c \geq 2$$

- On fait la même chose pour $T(3)$...
- ... et on obtient que c doit être ≥ 2.

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\frac{n}{2}\right) \leq c\frac{n}{2} \log \frac{n}{2}$$

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\tfrac{n}{2}\right) \leq c\tfrac{n}{2} \log \tfrac{n}{2}$$

- On vérifie que c'est aussi le cas pour $x = n$ en substituant la formule pour $T(x)$ dans son expression récursive :

$$T(n) \quad = \quad 2T\left(\tfrac{n}{2}\right) + n$$

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\tfrac{n}{2}\right) \leq c\tfrac{n}{2} \log \tfrac{n}{2}$$

- On vérifie que c'est aussi le cas pour $x = n$ en substituant la formule pour $T(x)$ dans son expression récursive :

$$\begin{array}{rcl} T(n) & = & 2T\left(\frac{n}{2}\right) + n \\ & \leq & 2c\frac{n}{2} \log(\frac{n}{2}) + n \end{array}$$

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\tfrac{n}{2}\right) \leq c\tfrac{n}{2} \log \tfrac{n}{2}$$

- On vérifie que c'est aussi le cas pour $x = n$ en substituant la formule pour $T(x)$ dans son expression récursive :

$$\begin{array}{rcl} T(n) & = & 2T\left(\frac{n}{2}\right) + n \\ & \leq & 2c\frac{n}{2} \log(\frac{n}{2}) + n \\ & \leq & cn \log \frac{n}{2} + n \end{array}$$

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\tfrac{n}{2}\right) \leq c\tfrac{n}{2} \log \tfrac{n}{2}$$

- On vérifie que c'est aussi le cas pour $x = n$ en substituant la formule pour $T(x)$ dans son expression récursive :

$$\begin{aligned} T(n) &= 2T\left(\tfrac{n}{2}\right) + n \\ &\leq 2c\tfrac{n}{2} \log(\tfrac{n}{2}) + n \\ &\leq cn \log \tfrac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \end{aligned}$$

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\tfrac{n}{2}\right) \leq c\tfrac{n}{2} \log \tfrac{n}{2}$$

- On vérifie que c'est aussi le cas pour $x = n$ en substituant la formule pour $T(x)$ dans son expression récursive :

$$\begin{aligned} T(n) &= 2T\left(\tfrac{n}{2}\right) + n \\ &\leq 2c\tfrac{n}{2} \log(\tfrac{n}{2}) + n \\ &\leq cn \log \tfrac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \end{aligned}$$

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\tfrac{n}{2}\right) \leq c\tfrac{n}{2} \log \tfrac{n}{2}$$

- On vérifie que c'est aussi le cas pour $x = n$ en substituant la formule pour $T(x)$ dans son expression récursive :

$$\begin{array}{rcll} T(n) & = & 2T\left(\frac{n}{2}\right) + n & \\ & \leq & 2c\frac{n}{2}\log(\frac{n}{2}) + n & \\ & \leq & cn \log \frac{n}{2} + n & \\ & = & cn \log n - cn \log 2 + n & \\ & = & cn \log n - cn + n & \\ & \leq & cn \log n & (\text{pour } c \geq 1) \end{array}$$

- On a pris $c \geq 2$, pour satisfaire les conditions initiales $T(2)$ et $T(3)$

Diviser pour régner : `TriFusion`

Exemple

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n^1) & \text{si } n > 1 \end{cases}$$

Diviser pour régner : `TriFusion`

Exemple

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
si $|L| \leq 1$ **alors**
    retourner L;
sinon
    $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
    $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
    $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
    retourner `Fusion`(L_l, L_r);

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ aT\left(\frac{n}{b}\right) + \Theta(n^d) & \text{si } n > 1 \end{cases}$$

- Trifusion : $a = 2$, $b = 2$, $d = 1$

- L'algorithme découpe la donnée en a sous-problèmes de taille $\frac{n}{b}$, les résout récursivement et rassemble les réponses en $\Theta(n^d)$

Diviser pour régner : `RechBin`

Exemple

Algorithme : `RechBin` (L)

Données : tableau trié L contenant e
Résultat : la position de e dans L

$m \leftarrow \left\lfloor \frac{|L|}{2} \right\rfloor$;
si $L[m] = e$ **alors**
  retourner m
sinon si $L[m] < e$ **alors**
  retourner `RechBin(`$L[m+1:]$`)`
sinon
  retourner `RechBin(`$L[:m]$`)`

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ aT\left(\frac{n}{b}\right) + \Theta(n^d) & \text{si } n > 1 \end{cases}$$

- L'algorithme découpe la donnée en a sous-problèmes de taille $\frac{n}{b}$, les résout récursivement et rassemble les réponses en $\Theta(n^d)$

Diviser pour régner : `RechBin`

Exemple

Algorithme : `RechBin` (L)

Données : tableau trié L contenant e
Résultat : la position de e dans L

$m \leftarrow \left\lfloor \frac{|L|}{2} \right\rfloor$;
si $L[m] = e$ **alors**
  retourner m
sinon si $L[m] < e$ **alors**
  retourner `RechBin`$(L[m+1:])$
sinon
  retourner `RechBin`$(L[:m])$

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 1T\left(\frac{n}{2}\right) + \Theta(n^0) & \text{si } n > 1 \end{cases}$$

- Recherche Binaire : $a = 1$, $b = 2$, $d = 0$

- L'algorithme découpe la donnée en a sous-problèmes de taille $\frac{n}{b}$, les résout récursivement et rassemble les réponses en $\Theta(n^d)$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Tri fusion :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Tri fusion :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $a = 2,\ b = 2,\ d = 1,$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1	Si $d > \log_b a$, $T(n) = \Theta(n^d)$	complexité dominée par le coût de fusion
2	Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$	complexité dominée par le coût du sous-problème
3	Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$	pas de domination

- Tri fusion :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $a = 2,\ b = 2,\ d = 1,\ \log_2 2 = 1 = d$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Tri fusion :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $a = 2,\ b = 2,\ d = 1,\ \log_2 2 = 1 = d$

  On est donc dans le 3ème cas et la complexité en $\Theta(n \log n)$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $a = 1$,

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $a = 1,\ b = 2,$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $a = 1,\ b = 2,\ d = 0,$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $a = 1,\ b = 2,\ d = 0,\ \log_2 1 = 0 = d$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $a = 1,\ b = 2,\ d = 0,\ \log_2 1 = 0 = d$

  On est donc dans le cas 3 et la complexité en $\Theta(\log n)$