Algorithmique des graphes

9 — Techniques algorithmiques

Anthony Labarre

7 avril 2021

Techniques algorithmiques

- Une grande partie des algorithmes vus dans les chapitres précédents peuvent être classés en plusieurs catégories :

Techniques algorithmiques

- Une grande partie des algorithmes vus dans les chapitres précédents peuvent être classés en plusieurs catégories :
  1. les algorithmes **gloutons** (Dijkstra, Prim, Kruskal, ...) ;

Techniques algorithmiques

- Une grande partie des algorithmes vus dans les chapitres précédents peuvent être classés en plusieurs catégories :
  1. les algorithmes **gloutons** (Dijkstra, Prim, Kruskal, . . .) ;
  2. les algorithmes “**diviser pour régner**” ;

Techniques algorithmiques

- Une grande partie des algorithmes vus dans les chapitres précédents peuvent être classés en plusieurs catégories :
  1. les algorithmes **gloutons** (Dijkstra, Prim, Kruskal, ...) ;
  2. les algorithmes "**diviser pour régner**" ;
  3. les algorithmes de **programmation dynamique** (Bellman-Ford, Floyd-Warshall, ...).

Techniques algorithmiques

- Une grande partie des algorithmes vus dans les chapitres précédents peuvent être classés en plusieurs catégories :
  1. les algorithmes **gloutons** (Dijkstra, Prim, Kruskal, . . .) ;
  2. les algorithmes "**diviser pour régner**" ;
  3. les algorithmes de **programmation dynamique** (Bellman-Ford, Floyd-Warshall, . . .).
- On va examiner plus en profondeur les principes de ces algorithmes ;

Points communs

- Les trois techniques que l'on vient de mentionner ont un point commun ;

Points communs

- Les trois techniques que l'on vient de mentionner ont un point commun ;
- Il s'agit de la subdivision du problème d'entrée en **sous-problèmes** ;

Points communs

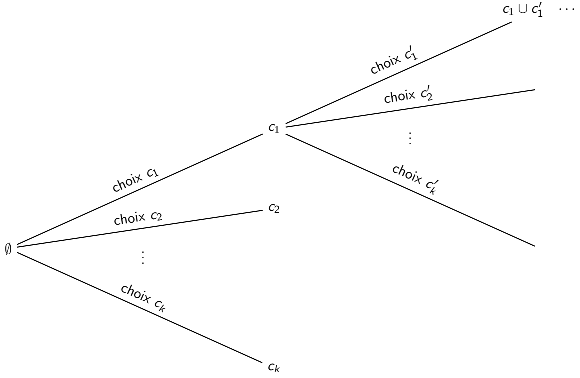
- Les trois techniques que l'on vient de mentionner ont un point commun ;
- Il s'agit de la subdivision du problème d'entrée en **sous-problèmes** ;
- Par "sous-problème", on veut dire que l'on veut résoudre le même problème que celui de départ, mais sur une instance plus petite qui est un morceau de celle de départ ;

Points communs

- Les trois techniques que l'on vient de mentionner ont un point commun ;
- Il s'agit de la subdivision du problème d'entrée en **sous-problèmes** ;
- Par "sous-problème", on veut dire que l'on veut résoudre le même problème que celui de départ, mais sur une instance plus petite qui est un morceau de celle de départ ;
- Les techniques vont différer selon la manière dont on résoud les sous-problèmes et dont on exploite leurs solutions ;

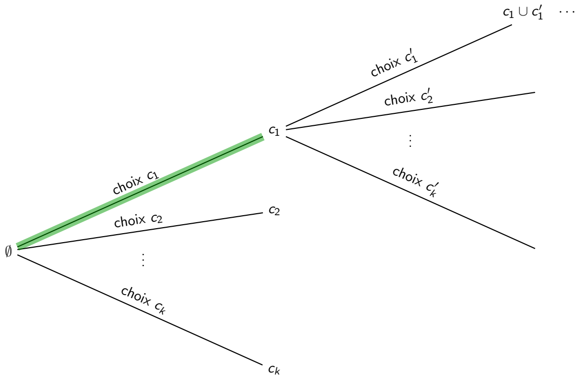
Principe

L'approche gloutonne se résume en une seule phrase : à chaque étape, on fait le “meilleur” choix (à définir bien sûr).



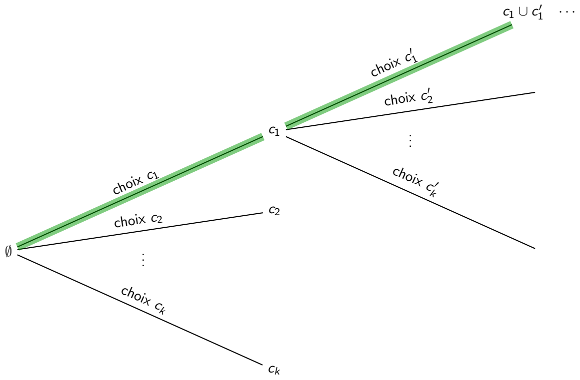
Principe

L'approche gloutonne se résume en une seule phrase : à chaque étape, on fait le “meilleur” choix (à définir bien sûr).



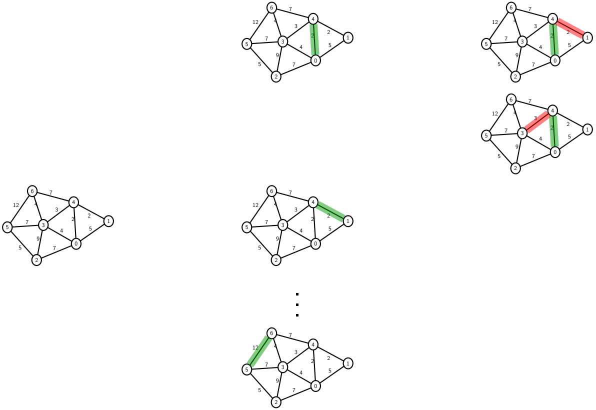
Principe

L'approche gloutonne se résume en une seule phrase : à chaque étape, on fait le "meilleur" choix (à définir bien sûr).



Exemple : forêt couvrante de poids minimum (Kruskal)

- Le poids de la forêt est la somme des poids de ses arêtes ;
- Stratégie gloutonne : privilégier les arêtes de poids minimum ;
- Bien entendu, on n'accepte que les arêtes *valides* ;



Exemple : plus courts chemins (Dijkstra)

- Le schéma glouton est moins évident pour l'algorithme de Dijkstra, mais la philosophie est la même ;
- Le but est de découvrir les plus courts chemins d'un sommet s à tous les autres ;
  - on parcourt les sommets par proximité (distance estimée minimale à s) ;
  - à chaque nouveau sommet découvert, on profite des raccourcis passant par ce sommet pour améliorer nos estimations ;

Caractéristiques des algorithmes gloutons

- Les algorithmes gloutons effectuent des choix **localement** optimaux dans l'espoir d'obtenir une solution **globalement** optimale ;

Caractéristiques des algorithmes gloutons

- Les algorithmes gloutons effectuent des choix **localement** optimaux dans l'espoir d'obtenir une solution **globalement** optimale ;
- Ils ne remettent pas en question les choix effectués aux étapes antérieures ;

Caractéristiques des algorithmes gloutons

- Les algorithmes gloutons effectuent des choix **localement** optimaux dans l'espoir d'obtenir une solution **globalement** optimale ;
- Ils ne remettent pas en question les choix effectués aux étapes antérieures ;
- Pour qu'ils fonctionnent, il faut donc que le problème possède une structure particulière ;

Caractéristiques des algorithmes gloutons

- Les algorithmes gloutons effectuent des choix **localement** optimaux dans l'espoir d'obtenir une solution **globalement** optimale ;
- Ils ne remettent pas en question les choix effectués aux étapes antérieures ;
- Pour qu'ils fonctionnent, il faut donc que le problème possède une structure particulière ;
- Pour en savoir plus : voir [1], chapitre sur les *matroïdes* ;

Preuve d'optimalité

- Pour prouver l'optimalité d'un algorithme glouton (quand c'est bien le cas), on peut procéder comme suit :

Preuve d'optimalité

- Pour prouver l'optimalité d'un algorithme glouton (quand c'est bien le cas), on peut procéder comme suit :
  1. soit T une solution optimale différente de la solution gloutonne S ;

Preuve d'optimalité

- Pour prouver l'optimalité d'un algorithme glouton (quand c'est bien le cas), on peut procéder comme suit :
  1. soit T une solution optimale différente de la solution gloutonne S ;
  2. on cherche la "première" différence entre S et T (par exemple en position i) ;

Preuve d'optimalité

- Pour prouver l'optimalité d'un algorithme glouton (quand c'est bien le cas), on peut procéder comme suit :

  1. soit T une solution optimale différente de la solution gloutonne S ;
  2. on cherche la "première" différence entre S et T (par exemple en position i) ;
  3. on montre que l'on peut remplacer la décision prise à l'étape i dans T par une décision gloutonne sans dégrader la qualité de T ;

Preuve d'optimalité

- Pour prouver l'optimalité d'un algorithme glouton (quand c'est bien le cas), on peut procéder comme suit :

  1. soit T une solution optimale différente de la solution gloutonne S ;
  2. on cherche la "première" différence entre S et T (par exemple en position i) ;
  3. on montre que l'on peut remplacer la décision prise à l'étape i dans T par une décision gloutonne sans dégrader la qualité de T ;

- Si l'on y parvient, la stratégie gloutonne est correcte, puisqu'on peut transformer toute solution optimale en une solution gloutonne sans la dégrader ;

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

□

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

Soit S une solution gloutonne (Prim ou Kruskal),

$$S = \quad e_1 \quad e_2 \quad \cdots \quad e_{i-1} \quad e_i \quad e_{i+1} \quad \cdots \quad e_n$$

□

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

Soit S une solution gloutonne (Prim ou Kruskal), T une solution optimale de même longueur,

$$\begin{array}{lcccccccc} S = & e_1 & e_2 & \cdots & e_{i-1} & e_i & e_{i+1} & \cdots & e_n \\ T = & f_1 & f_2 & \cdots & f_{i-1} & f_i & f_{i+1} & \cdots & f_n \end{array}$$

□

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

Soit S une solution gloutonne (Prim ou Kruskal), T une solution optimale de même longueur, et i la position de la première différence entre S et T :

$$\begin{array}{rcccccccc} S = & e_1 & e_2 & \cdots & e_{i-1} & e_i & e_{i+1} & \cdots & e_n \\ T = & f_1 & f_2 & \cdots & f_{i-1} & f_i & f_{i+1} & \cdots & f_n \\ = & e_1 & e_2 & \cdots & e_{i-1} & f_i & f_{i+1} & \cdots & f_n \end{array}$$

$\square$

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

Soit S une solution gloutonne (Prim ou Kruskal), T une solution optimale de même longueur, et i la position de la première différence entre S et T :

$$\begin{array}{rcccccccc} S = & e_1 & e_2 & \cdots & e_{i-1} & e_i & e_{i+1} & \cdots & e_n \\ T = & f_1 & f_2 & \cdots & f_{i-1} & f_i & f_{i+1} & \cdots & f_n \\ = & e_1 & e_2 & \cdots & e_{i-1} & f_i & f_{i+1} & \cdots & f_n \end{array}$$

Remplaçons f_i par e_i dans T ; on obtient

$$\begin{array}{rcccccccc} T' = & e_1 & e_2 & \cdots & e_{i-1} & e_i & f_{i+1} & \cdots & f_n \end{array}$$

$\square$

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

Soit S une solution gloutonne (Prim ou Kruskal), T une solution optimale de même longueur, et i la position de la première différence entre S et T :

$$\begin{array}{rcccccccc} S = & e_1 & e_2 & \cdots & e_{i-1} & e_i & e_{i+1} & \cdots & e_n \\ T = & f_1 & f_2 & \cdots & f_{i-1} & f_i & f_{i+1} & \cdots & f_n \\ = & e_1 & e_2 & \cdots & e_{i-1} & f_i & f_{i+1} & \cdots & f_n \end{array}$$

Remplaçons f_i par e_i dans T ; on obtient

$$\begin{array}{rcccccccc} T' = & e_1 & e_2 & \cdots & e_{i-1} & e_i & f_{i+1} & \cdots & f_n \end{array}$$

Le coût de T' est $w(T') = w(T) - w(f_i) + w(e_i)$

□

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

Soit S une solution gloutonne (Prim ou Kruskal), T une solution optimale de même longueur, et i la position de la première différence entre S et T :

$$\begin{array}{rcccccccc} S = & e_1 & e_2 & \cdots & e_{i-1} & e_i & e_{i+1} & \cdots & e_n \\ T = & f_1 & f_2 & \cdots & f_{i-1} & f_i & f_{i+1} & \cdots & f_n \\ = & e_1 & e_2 & \cdots & e_{i-1} & f_i & f_{i+1} & \cdots & f_n \end{array}$$

Remplaçons f_i par e_i dans T ; on obtient

$$\begin{array}{rcccccccc} T' = & e_1 & e_2 & \cdots & e_{i-1} & e_i & f_{i+1} & \cdots & f_n \end{array}$$

Le coût de T' est $w(T') = w(T) - w(f_i) + w(e_i) \leq w(T)$ puisque e_i est de poids minimum parmi les arêtes valides.

□

Correction de Prim / Kruskal

Proposition 1

Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration.

Soit S une solution gloutonne (Prim ou Kruskal), T une solution optimale de même longueur, et i la position de la première différence entre S et T :

$$\begin{array}{rcccccccc} S = & e_1 & e_2 & \cdots & e_{i-1} & e_i & e_{i+1} & \cdots & e_n \\ T = & f_1 & f_2 & \cdots & f_{i-1} & f_i & f_{i+1} & \cdots & f_n \\ = & e_1 & e_2 & \cdots & e_{i-1} & f_i & f_{i+1} & \cdots & f_n \end{array}$$

Remplaçons f_i par e_i dans T ; on obtient

$$\begin{array}{rcccccccc} T' = & e_1 & e_2 & \cdots & e_{i-1} & e_i & f_{i+1} & \cdots & f_n \end{array}$$

Le coût de T' est $w(T') = w(T) - w(f_i) + w(e_i) \leq w(T)$ puisque e_i est de poids minimum parmi les arêtes valides. Il nous suffit donc de recommencer jusqu'à ce que T devienne S ; aucune transformation n'augmente le coût de T, donc S est optimale puisque $w(S) \leq w(T)$. □

Plus courte super-séquence commune

- Pour éviter de croire que la stratégie gloutonne marche toujours, examinons un problème où elle semble naturelle mais échoue ;

Plus courte super-séquence commune

- Pour éviter de croire que la stratégie gloutonne marche toujours, examinons un problème où elle semble naturelle mais échoue ;
- Soit $S = (s_1, s_2, \ldots, s_p)$ et $T = (t_1, t_2, \ldots, t_q)$ avec $q \geq p$; s'il existe une bijection entre S et une sous-séquence de T préservant l'ordre des éléments de S, alors S est une **sous-séquence** de T, et T est une **super-séquence** de S.

Plus courte super-séquence commune

- Pour éviter de croire que la stratégie gloutonne marche toujours, examinons un problème où elle semble naturelle mais échoue ;
- Soit $S = (s_1, s_2, \ldots, s_p)$ et $T = (t_1, t_2, \ldots, t_q)$ avec $q \geq p$; s'il existe une bijection entre S et une sous-séquence de T préservant l'ordre des éléments de S, alors S est une **sous-séquence** de T, et T est une **super-séquence** de S.

Exemple 1

BONJOUR est une sous-séquence de BONNE JOURNÉE

Plus courte super-séquence commune

- Le problème de la **plus courte super-séquence commune** demande de trouver une super-séquence commune à un ensemble de séquences données qui soit la plus courte possible.

Plus courte super-séquence commune

- Le problème de la **plus courte super-séquence commune** demande de trouver une super-séquence commune à un ensemble de séquences données qui soit la plus courte possible.
- Une approche gloutonne possible : tant qu'on a plus de deux chaînes, fusionner les deux chaînes dont la superséquence commune optimale est la plus “compacte” possible ;

Plus courte super-séquence commune

- Le problème de la **plus courte super-séquence commune** demande de trouver une super-séquence commune à un ensemble de séquences données qui soit la plus courte possible.
- Une approche gloutonne possible : tant qu'on a plus de deux chaînes, fusionner les deux chaînes dont la superséquence commune optimale est la plus "compacte" possible ;
- Comment mesurer cette compacité ? On peut choisir de prendre le nombre de caractères supplémentaires nécessaires.

Plus courte super-séquence commune

- Le problème de la **plus courte super-séquence commune** demande de trouver une super-séquence commune à un ensemble de séquences données qui soit la plus courte possible.
- Une approche gloutonne possible : tant qu'on a plus de deux chaînes, fusionner les deux chaînes dont la superséquence commune optimale est la plus "compacte" possible ;
- Comment mesurer cette compacité ? On peut choisir de prendre le nombre de caractères supplémentaires nécessaires.

Exemple 2

GRAPHES (7 lettres) et ALGORITHMES (11 lettres) se fusionnent de manière optimale en ALGORIAPTHMES (13 lettres), au prix de $13 - 11 = 2$ lettres supplémentaires.

Plus courte super-séquence commune

Exemple 3 (approche gloutonne)

LES

ALGOS

GLOUTONS

SONT

NULS

Plus courte super-séquence commune

Exemple 3 (approche gloutonne)

LES → NULES

ALGOS

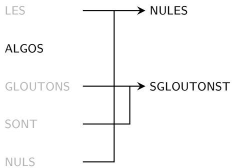
GLOUTONS

SONT

NULS

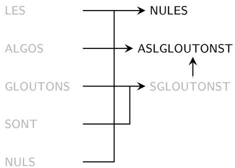
Plus courte super-séquence commune

Exemple 3 (approche gloutonne)



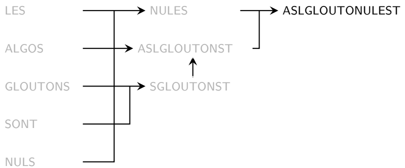
Plus courte super-séquence commune

Exemple 3 (approche gloutonne)



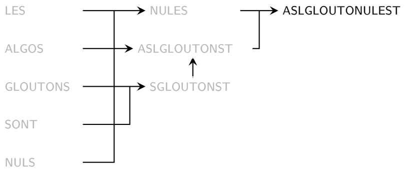
Plus courte super-séquence commune

Exemple 3 (approche gloutonne)



Plus courte super-séquence commune

Exemple 3 (approche gloutonne)



La solution gloutonne est de longueur 15, mais il existe une solution de taille 14 :

A	S	L	E	G	L	O	N	U	T	L	O	N	S
		L	E										S
A		L		G		O							S
				G	L	O		U	T		O	N	S
	S					O	N		T				
							N	U		L			S

Bilan des algorithmes gloutons

- L'approche gloutonne donne rarement des solutions optimales ;

Bilan des algorithmes gloutons

- L'approche gloutonne donne rarement des solutions optimales ;
- Il s'agit néanmoins d'une approche utile car :

Bilan des algorithmes gloutons

- L'approche gloutonne donne rarement des solutions optimales ;
- Il s'agit néanmoins d'une approche utile car :
  - elle est intuitive ;

Bilan des algorithmes gloutons

- L'approche gloutonne donne rarement des solutions optimales ;
- Il s'agit néanmoins d'une approche utile car :
  - elle est intuitive ;
  - elle est simple à implémenter ;

Bilan des algorithmes gloutons

- L'approche gloutonne donne rarement des solutions optimales ;
- Il s'agit néanmoins d'une approche utile car :
  - elle est intuitive ;
  - elle est simple à implémenter ;
  - elle peut donner de bons résultats approximatifs ;

Bilan des algorithmes gloutons

- L'approche gloutonne donne rarement des solutions optimales ;
- Il s'agit néanmoins d'une approche utile car :
  - elle est intuitive ;
  - elle est simple à implémenter ;
  - elle peut donner de bons résultats approximatifs ;
- Le dernier point est important quand on a affaire à des problèmes difficiles (NP-complets) ;

Introduction rapide et approximative aux classes de complexité

- On a :

Introduction rapide et approximative aux classes de complexité

- On a :
  - P : la classe des problèmes solubles en temps polynomial (en la taille de l'entrée) ;

Introduction rapide et approximative aux classes de complexité

- On a :
  - P : la classe des problèmes solubles en temps polynomial (en la taille de l'entrée) ;
  - NP : la classe des problèmes vérifiables en temps polynomial ;

Introduction rapide et approximative aux classes de complexité

- On a :
  - P : la classe des problèmes solubles en temps polynomial (en la taille de l'entrée) ;
  - NP : la classe des problèmes vérifiables en temps polynomial ;
  - les problèmes NP-complets et NP-durs : on conjecture qu'ils ne sont pas dans P ;

Introduction rapide et approximative aux classes de complexité

- On a :
  - P : la classe des problèmes solubles en temps polynomial (en la taille de l'entrée) ;
  - NP : la classe des problèmes vérifiables en temps polynomial ;
  - les problèmes NP-complets et NP-durs : on conjecture qu'ils ne sont pas dans P ;
- La tête du problème P $\stackrel{?}{=}$ NP est mise à prix à \$ 1,000,000 ;

Introduction rapide et approximative aux classes de complexité

- On a :
  - P : la classe des problèmes solubles en temps polynomial (en la taille de l'entrée) ;
  - NP : la classe des problèmes vérifiables en temps polynomial ;
  - les problèmes NP-complets et NP-durs : on conjecture qu'ils ne sont pas dans P ;
- La tête du problème $P \stackrel{?}{=} NP$ est mise à prix à \$ 1,000,000 ;
- Si un problème d'optimisation est "difficile", on doit généralement choisir entre :

Introduction rapide et approximative aux classes de complexité

- On a :
  - P : la classe des problèmes solubles en temps polynomial (en la taille de l'entrée) ;
  - NP : la classe des problèmes vérifiables en temps polynomial ;
  - les problèmes NP-complets et NP-durs : on conjecture qu'ils ne sont pas dans P ;
- La tête du problème $P \stackrel{?}{=} NP$ est mise à prix à \$ 1,000,000 ;
- Si un problème d'optimisation est "difficile", on doit généralement choisir entre :
  1. trouver une solution rapidement (mais sous-optimale) ;

Introduction rapide et approximative aux classes de complexité

- On a :
  - P : la classe des problèmes solubles en temps polynomial (en la taille de l'entrée) ;
  - NP : la classe des problèmes vérifiables en temps polynomial ;
  - les problèmes NP-complets et NP-durs : on conjecture qu'ils ne sont pas dans P ;
- La tête du problème $P \stackrel{?}{=} NP$ est mise à prix à \$ 1,000,000 ;
- Si un problème d'optimisation est "difficile", on doit généralement choisir entre :
  1. trouver une solution rapidement (mais sous-optimale) ;
  2. trouver une solution optimale (mais lentement) ;

Diviser pour régner

- L'approche "diviser pour régner" comporte trois ingrédients :

Diviser pour régner

- L'approche "diviser pour régner" comporte trois ingrédients :
  1. **diviser** : découper le problème en sous-problèmes ;

Diviser pour régner

- L'approche "diviser pour régner" comporte trois ingrédients :
  1. **diviser** : découper le problème en sous-problèmes ;
  2. **régner** : résoudre les sous-problèmes récursivement (ou directement s'ils sont de taille suffisamment petite) ;

Diviser pour régner

- L'approche "diviser pour régner" comporte trois ingrédients :
  1. **diviser** : découper le problème en sous-problèmes ;
  2. **régner** : résoudre les sous-problèmes récursivement (ou directement s'ils sont de taille suffisamment petite) ;
  3. **combiner** : combiner les solutions des sous-problèmes pour obtenir la solution au problème de départ.

Diviser pour régner

- L'approche "diviser pour régner" comporte trois ingrédients :
  1. **diviser** : découper le problème en sous-problèmes ;
  2. **régner** : résoudre les sous-problèmes récursivement (ou directement s'ils sont de taille suffisamment petite) ;
  3. **combiner** : combiner les solutions des sous-problèmes pour obtenir la solution au problème de départ.
- L'efficacité de cette approche dépend fortement des complexités de ces trois phases ; il faut que :

Diviser pour régner

- L'approche "diviser pour régner" comporte trois ingrédients :
  1. **diviser** : découper le problème en sous-problèmes ;
  2. **régner** : résoudre les sous-problèmes récursivement (ou directement s'ils sont de taille suffisamment petite) ;
  3. **combiner** : combiner les solutions des sous-problèmes pour obtenir la solution au problème de départ.
- L'efficacité de cette approche dépend fortement des complexités de ces trois phases ; il faut que :
  1. le découpage soit bien choisi ;

Diviser pour régner

- L'approche "diviser pour régner" comporte trois ingrédients :
  1. **diviser** : découper le problème en sous-problèmes ;
  2. **régner** : résoudre les sous-problèmes récursivement (ou directement s'ils sont de taille suffisamment petite) ;
  3. **combiner** : combiner les solutions des sous-problèmes pour obtenir la solution au problème de départ.
- L'efficacité de cette approche dépend fortement des complexités de ces trois phases ; il faut que :
  1. le découpage soit bien choisi ;
  2. combiner deux sous-solutions prenne moins de temps que de calculer la solution sans procéder à la division.

Recherche d'un élément dans une séquence triée

- Approche triviale : $O(n)$;

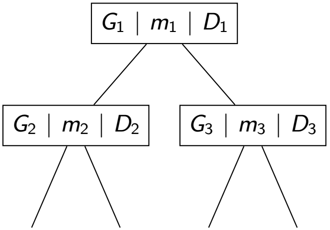
Recherche d'un élément dans une séquence triée

- Approche triviale : $O(n)$;
- Approche diviser pour régner : $O(\log n)$ (dichotomie)

Recherche d'un élément dans une séquence triée

- Approche triviale : $O(n)$;
- Approche diviser pour régner : $O(\log n)$ (dichotomie)

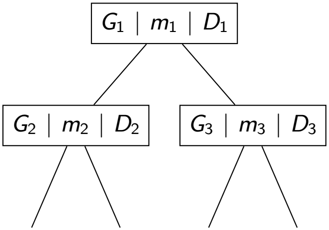
1. **diviser** : diviser l'intervalle de recherche en deux parties (presque) égales ;



Recherche d'un élément dans une séquence triée

- Approche triviale : $O(n)$;
- Approche diviser pour régner : $O(\log n)$ (dichotomie)

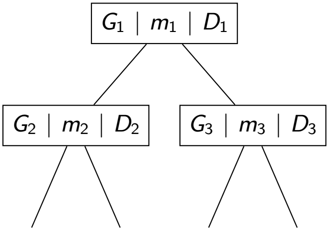
1. **diviser** : diviser l'intervalle de recherche en deux parties (presque) égales ;
2. **régner** : renvoyer la position du milieu ou relancer la recherche récursivement dans la "bonne" partie ;



Recherche d'un élément dans une séquence triée

- Approche triviale : $O(n)$;
- Approche diviser pour régner : $O(\log n)$ (dichotomie)

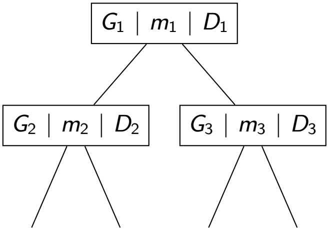
1. **diviser** : diviser l'intervalle de recherche en deux parties (presque) égales ;
2. **régner** : renvoyer la position du milieu ou relancer la recherche récursivement dans la "bonne" partie ;
3. **combiner** : renvoyer le premier résultat fructueux (ou NIL).



Recherche d'un élément dans une séquence triée

- Approche triviale : $O(n)$;
- Approche diviser pour régner : $O(\log n)$ (dichotomie)

1. **diviser** : diviser l'intervalle de recherche en deux parties (presque) égales ;
2. **régner** : renvoyer la position du milieu ou relancer la recherche récursivement dans la "bonne" partie ;
3. **combiner** : renvoyer le premier résultat fructueux (ou NIL).



$O(\log n)$ découpes, toutes les autres opérations sont en $O(1) \Rightarrow O(\log n)$ au total.

Tri d'une séquence

- Algorithmes basiques : $O(n^2)$

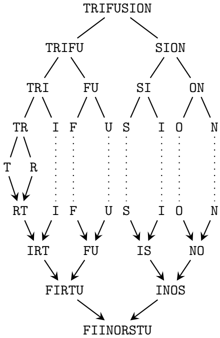
Tri d'une séquence

- Algorithmes basiques : $O(n^2)$
- Approche diviser pour régner : $O(n \log n)$ (tri fusion)

Tri d'une séquence

- Algorithmes basiques : $O(n^2)$
- Approche diviser pour régner : $O(n \log n)$ (tri fusion)

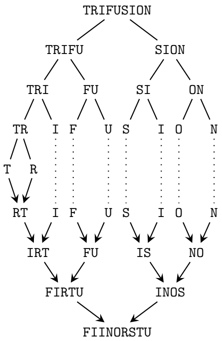
1. **diviser** : diviser la séquence en deux parties (presque) égales ;



Tri d'une séquence

- Algorithmes basiques : $O(n^2)$
- Approche diviser pour régner : $O(n \log n)$ (tri fusion)

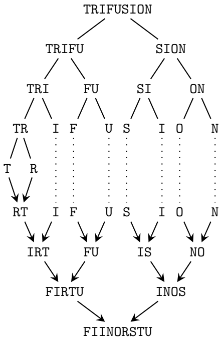
1. **diviser** : diviser la séquence en deux parties (presque) égales ;
2. **régner** : ne rien faire (longueur ≤ 1) ou trier récursivement les deux sous-séquences ;



Tri d'une séquence

- Algorithmes basiques : $O(n^2)$
- Approche diviser pour régner : $O(n \log n)$ (tri fusion)

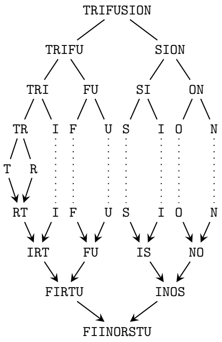
1. **diviser** : diviser la séquence en deux parties (presque) égales ;
2. **régner** : ne rien faire (longueur ≤ 1) ou trier récursivement les deux sous-séquences ;
3. **combiner** : fusionner les deux sous-parties triées en une séquence triée.



Tri d'une séquence

- Algorithmes basiques : $O(n^2)$
- Approche diviser pour régner : $O(n \log n)$ (tri fusion)

1. **diviser** : diviser la séquence en deux parties (presque) égales ;
2. **régner** : ne rien faire (longueur ≤ 1) ou trier récursivement les deux sous-séquences ;
3. **combiner** : fusionner les deux sous-parties triées en une séquence triée.



Hauteur de l'arbre : $O(\log n)$, fusions en $O(n) \Rightarrow O(n \log n)$ au total.

Applications de l'approche diviser pour régner

- Pour que l'approche "diviser pour régner" fonctionne, il faut :

Applications de l'approche diviser pour régner

- Pour que l'approche "diviser pour régner" fonctionne, il faut :
  - que le problème s'y prête ;

Applications de l'approche diviser pour régner

- Pour que l'approche "diviser pour régner" fonctionne, il faut :
  - que le problème s'y prête ;
  - que l'on ait un moyen efficace de combiner les solutions ;

Applications de l'approche diviser pour régner

- Pour que l'approche "diviser pour régner" fonctionne, il faut :
  - que le problème s'y prête ;
  - que l'on ait un moyen efficace de combiner les solutions ;
- Est-ce que ça fonctionne toujours ?

Applications de l'approche diviser pour régner

- Pour que l'approche "diviser pour régner" fonctionne, il faut :
  - que le problème s'y prête ;
  - que l'on ait un moyen efficace de combiner les solutions ;
- Est-ce que ça fonctionne toujours ?
  - non : voir la tentative d'algorithme récursif pour le calcul d'un ACPM vue en TD ;

Bilan de l'approche diviser pour régner

- L'approche "diviser pour régner" est un peu moins naturelle que les algorithmes gloutons ;

Bilan de l'approche diviser pour régner

- L'approche "diviser pour régner" est un peu moins naturelle que les algorithmes gloutons ;
- En général, on y pense plutôt quand on a déjà un algorithme pour résoudre le problème donné mais qu'on veut en obtenir un plus efficace ;

Bilan de l'approche diviser pour régner

- L'approche "diviser pour régner" est un peu moins naturelle que les algorithmes gloutons ;
- En général, on y pense plutôt quand on a déjà un algorithme pour résoudre le problème donné mais qu'on veut en obtenir un plus efficace ;
- Difficultés (liées) :

Bilan de l'approche diviser pour régner

- L'approche "diviser pour régner" est un peu moins naturelle que les algorithmes gloutons ;
- En général, on y pense plutôt quand on a déjà un algorithme pour résoudre le problème donné mais qu'on veut en obtenir un plus efficace ;
- Difficultés (liées) :
  - trouver "le" bon découpage ;

Bilan de l'approche diviser pour régner

- L'approche "diviser pour régner" est un peu moins naturelle que les algorithmes gloutons ;
- En général, on y pense plutôt quand on a déjà un algorithme pour résoudre le problème donné mais qu'on veut en obtenir un plus efficace ;
- Difficultés (liées) :
  - trouver "le" bon découpage ;
  - et "la" manière efficace de combiner les sous-solutions ;

Bibliographie

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Introduction to Algorithms.
MIT Press, 3ème edition, 2009.