Décomposition en sous-problèmes

16 mai 2022

Table of Contents

Diviser pour régner : exemple

L'exponentiation sur les entiers (récursive) naïve peut se traduire comme suit

```
let rec puissance x = function
    0->1
   |n->x*puissance x (n-1);;
```

Diviser pour régner : exemple

L'exponentiation sur les entiers (récursive) naïve peut se traduire comme suit

```
let rec puissance x = function
    0->1
   |n->x*puissance x (n-1);;
```

La complexité temporelle de ce calcul est linéaire. Peut-on faire mieux ?

Diviser pour régner : exemple

L'exponentiation sur les entiers (récursive) naïve peut se traduire comme suit

```
let rec puissance x = function
    0->1
   |n->x*puissance x (n-1);;
```

La complexité temporelle de ce calcul est linéaire. Peut-on faire mieux ?
Algorithme d'exponentiation rapide :

```
let rec puissance x = function
    0->1
   |1->x
   |n when n mod 2=0->puissance (x*x) (n/2)
   |n->x*puissance (x*x) (n/2);;
```

Diviser pour régner : exemple

La complexité de cet algorithme est logarithmique.

Diviser pour régner : exemple

La complexité de cet algorithme est logarithmique.
En effet, notons c_n le nombre de multiplications nécessaires (en fonction de n) :

- $c_0 = c_1 = 0$
- $c_{2k} = c_k + 1$
- $c_{2k+1} = c_k + 1$

Diviser pour régner : exemple

La complexité de cet algorithme est logarithmique.
En effet, notons c_n le nombre de multiplications nécessaires (en fonction de n) :

- $c_0 = c_1 = 0$
- $c_{2k} = c_k + 1$
- $c_{2k+1} = c_k + 1$

Le meilleur cas est celui d'une puissance de 2 ; le pire celui de nombres de la forme $2^p - 1$, dans les deux cas, logarithmique.

Paradigme diviser pour régner

Définition

Un algorithme **divise pour régner** (divide and conquer) lorsqu'il ramène la résolution d'un problème de taille n à la résolution de problèmes de taille plus petites *environ* égales à αn ($\alpha < 1$, souvent, $\alpha = \frac{1}{2}$)

Paradigme diviser pour régner

Définition

Un algorithme **divise pour régner** (divide and conquer) lorsqu'il ramène la résolution d'un problème de taille n à la résolution de problèmes de taille plus petites *environ* égales à αn ($\alpha < 1$, souvent, $\alpha = \frac{1}{2}$)

La solution est construite à l'aide des solutions des sous-problèmes.

Paradigme diviser pour régner

Définition

Un algorithme **divise pour régner** (divide and conquer) lorsqu'il ramène la résolution d'un problème de taille n à la résolution de problèmes de taille plus petites *environ* égales à αn ($\alpha < 1$, souvent, $\alpha = \frac{1}{2}$)

La solution est construite à l'aide des solutions des sous-problèmes. Exemple : tri fusion ; recherche dichotomique dans un tableau trié.

Complexité

Dans le cas où l'algorithme partage un problème de taille n en deux sous-problèmes de tailles respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$, la complexité respect la relation de récurrence de la forme suivante : $c_n = ac_{\lfloor \frac{n}{2} \rfloor} + bc_{\lceil \frac{n}{2} \rceil} + d_n$, avec $a + b \geq 1$.

Complexité

Dans le cas où l'algorithme partage un problème de taille n en deux sous-problèmes de tailles respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$, la complexité respect la relation de récurrence de la forme suivante : $c_n = ac_{\lfloor \frac{n}{2} \rfloor} + bc_{\lceil \frac{n}{2} \rceil} + d_n$, avec $a + b \geq 1$.

- On suppose que $d_n = \lambda n^k$. Si n est une puissance de 2 : m.q.
  - si $a + b < 2^k, c_n \sim \alpha 2^{k \log(n)}$
  - si $a + b = 2^k, c_n \sim \lambda \log n 2^{k \log(n)}$
  - si $a + b > 2^k, c_n \sim \beta (a + b)^{\log n}$

Complexité

Dans le cas où l'algorithme partage un problème de taille n en deux sous-problèmes de tailles respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$, la complexité respect la relation de récurrence de la forme suivante : $c_n = ac_{\lfloor \frac{n}{2} \rfloor} + bc_{\lceil \frac{n}{2} \rceil} + d_n$, avec $a + b \geq 1$.

- On suppose que $d_n = \lambda n^k$. Si n est une puissance de 2 : m.q.
  - si $a + b < 2^k, c_n \sim \alpha 2^{k \log(n)}$
  - si $a + b = 2^k, c_n \sim \lambda \log n 2^{k \log(n)}$
  - si $a + b > 2^k, c_n \sim \beta (a + b)^{\log n}$
- M.q. si $(d_n)_n$ est. croissante, alors $(c_n)_n$ l'est aussi.

Théorème maître

Proposition

Soit une suite définie par $c_0 \geq 0$ et $\forall n \geq 1, c_n = ac_{\lfloor \frac{n}{2} \rfloor} + bc_{\lceil \frac{n}{2} \rceil} + d_n$, avec $a+b > 1$, $(d_n)_n$ croissante, et $d_n = \Theta(n^k)$. On a :

- si $\log(a+b) < k, c_n = \Theta(n^k)$
- si $\log(a+b) = k, c_n = \Theta(n^k \log n)$
- si $\log(a+b) > k, c_n = \Theta(n^{\log(a+b)})$

Exemples

Tri fusion

Exemples

Tri fusion

On considère que la séparation en deux sous-listes et la fusion sont linéaires.

Exemples

Tri fusion

On considère que la séparation en deux sous-listes et la fusion sont linéaires. $c_n = c_{\lfloor \frac{n}{2} \rfloor} + c_{\lceil \frac{n}{2} \rceil} + \Theta(n)$ donc

Exemples

Tri fusion

On considère que la séparation en deux sous-listes et la fusion sont linéaires. $c_n = c_{\lfloor \frac{n}{2} \rfloor} + c_{\lceil \frac{n}{2} \rceil} + \Theta(n)$ donc $c_n = \Theta(n \log n)$

Distance minimale : recherche de deux points les plus proches dans un nuages de points.

Table of Contents

Cryptographie/cryptanalyse

Soit un algorithme A de cryptage. On pourrait envisager un algorithme A' consistant à crypter à l'aide de A successivement, et par deux clefs différentes. On s'attend à ce que le temps de décryptage soit élevé au carré.

Cryptographie/cryptanalyse

Soit un algorithme A de cryptage. On pourrait envisager un algorithme A' consistant à crypter à l'aide de A successivement, et par deux clefs différentes. On s'attend à ce que le temps de décryptage soit élevé au carré. La stratégie de **rencontre au milieu** choisit le compromis temps/mémoire.

Cryptographie/cryptanalyse

Soit un algorithme A de cryptage. On pourrait envisager un algorithme A' consistant à crypter à l'aide de A successivement, et par deux clefs différentes. On s'attend à ce que le temps de décryptage soit élevé au carré. La stratégie de **rencontre au milieu** choisit le compromis temps/mémoire. Cette stratégie consiste à trouver un texte clair M et sa version M' chiffrée par A'.

Cryptographie/cryptanalyse

Soit un algorithme A de cryptage. On pourrait envisager un algorithme A' consistant à crypter à l'aide de A successivement, et par deux clefs différentes. On s'attend à ce que le temps de décryptage soit élevé au carré. La stratégie de **rencontre au milieu** choisit le compromis temps/mémoire. Cette stratégie consiste à trouver un texte clair M et sa version M' chiffrée par A'. On code ensuite M par des clefs et par l'algorithme A, et on déchiffre M' par d'autre clefs (toujours utilisant A et non A', jusqu'à trouver le même message, codé de M, déchiffré de M' de l'autre.

Cryptographie/cryptanalyse

Soit un algorithme A de cryptage. On pourrait envisager un algorithme A' consistant à crypter à l'aide de A successivement, et par deux clefs différentes. On s'attend à ce que le temps de décryptage soit élevé au carré. La stratégie de **rencontre au milieu** choisit le compromis temps/mémoire. Cette stratégie consiste à trouver un texte clair M et sa version M' chiffrée par A'. On code ensuite M par des clefs et par l'algorithme A, et on déchiffre M' par d'autre clefs (toujours utilisant A et non A', jusqu'à trouver le même message, codé de M, déchiffré de M' de l'autre. On a multiplié (seulement) par deux le temps de recherche par rapport à l'algorithme A.

Cryptographie/cryptanalyse

Soit un algorithme A de cryptage. On pourrait envisager un algorithme A' consistant à crypter à l'aide de A successivement, et par deux clefs différentes. On s'attend à ce que le temps de décryptage soit élevé au carré. La stratégie de **rencontre au milieu** choisit le compromis temps/mémoire. Cette stratégie consiste à trouver un texte clair M et sa version M' chiffrée par A'. On code ensuite M par des clefs et par l'algorithme A, et on déchiffre M' par d'autre clefs (toujours utilisant A et non A', jusqu'à trouver le même message, codé de M, déchiffré de M' de l'autre. On a multiplié (seulement) par deux le temps de recherche par rapport à l'algorithme A.
Cette attaque rend ainsi le sur-chiffrement inutile.

Table of Contents

Exercices

Comptage du nombre d'inversions dans une liste.
Recherche dichotomique dans un tableau trié.
Dans les deux cas, comparer avec la version naïve.

Table of Contents

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions et en combinant les solutions de plusieurs sous-problèmes, non nécessairement disjoints.

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions et en combinant les solutions de plusieurs sous-problèmes, non nécessairement disjoints.

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions et en combinant les solutions de plusieurs sous-problèmes, non nécessairement disjoints. On n'énumère pas toutes les solutions, certaines sont rejetées avant même construction explicite, simplement parce qu'appartenant à un sous-ensemble dont on sait qu'il ne contient pas la solution cherchée.

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions et en combinant les solutions de plusieurs sous-problèmes, non nécessairement disjoints. On n'énumère pas toutes les solutions, certaines sont rejetées avant même construction explicite, simplement parce qu'appartenant à un sous-ensemble dont on sait qu'il ne contient pas la solution cherchée.

Définition

Principe d'optimalité de Bellman Un **chemin optimal** est formé de sous-chemins optimaux.

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions et en combinant les solutions de plusieurs sous-problèmes, non nécessairement disjoints. On n'énumère pas toutes les solutions, certaines sont rejetées avant même construction explicite, simplement parce qu'appartenant à un sous-ensemble dont on sait qu'il ne contient pas la solution cherchée.

Définition

Principe d'optimalité de Bellman Un **chemin optimal** est formé de sous-chemins optimaux.

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions et en combinant les solutions de plusieurs sous-problèmes, non nécessairement disjoints. On n'énumère pas toutes les solutions, certaines sont rejetées avant même construction explicite, simplement parce qu'appartenant à un sous-ensemble dont on sait qu'il ne contient pas la solution cherchée.

Définition

Principe d'optimalité de Bellman Un **chemin optimal** est formé de sous-chemins optimaux.

La programmation dynamique s'applique aux problèmes satisfaisant ce principe d'optimalité.

La programmation dynamique sert à résoudre des problèmes d'optimisation (trouver une solution optimale ou une valeur optimale) dans un grand ensemble de solutions et en combinant les solutions de plusieurs sous-problèmes, non nécessairement disjoints. On n'énumère pas toutes les solutions, certaines sont rejetées avant même construction explicite, simplement parce qu'appartenant à un sous-ensemble dont on sait qu'il ne contient pas la solution cherchée.

Définition

Principe d'optimalité de Bellman Un **chemin optimal** est formé de sous-chemins optimaux.

La programmation dynamique s'applique aux problèmes satisfaisant ce principe d'optimalité.
Exemples :

- trouver un plus court chemin (algorithme de Bellman-Ford et Floyd-Warshall)
- problème du sac -à- dos
- problème du rendu de monnaie **dans le cas général**
- recherche de la plus longue sous-suite strictement croissante dans une suite de nombres

Le principe d'optimalité de Bellman permet d'obtenir une formulation récursive, puis une résolution récursive d'un problème donné.

Le principe d'optimalité de Bellman permet d'obtenir une formulation récursive, puis une résolution récursive d'un problème donné.
Un algorithme de programmation dynamique procède généralement comme suit :

- caractériser ce qu'est une solution optimale
- définir (généralement par récurrence) une valeur optimale
- calculer la valeur d'une solution optimale (et garder en mémoire les valeurs des solutions optimales pour les problèmes plus petits)
- construire une solution optimale en utilisant les informations précalculées.

En pratique : après avoir obtenu une relation de récurrence liant la résolution globale à celle(s) de problèmes locaux , on initialise un tableau de taille précalculée, puis on le remplit "de bas en haut", en résolvant les problèmes locaux par taille croissante.

Le principe d'optimalité de Bellman permet d'obtenir une formulation récursive, puis une résolution récursive d'un problème donné.
Un algorithme de programmation dynamique procède généralement comme suit :

- caractériser ce qu'est une solution optimale
- définir (généralement par récurrence) une valeur optimale
- calculer la valeur d'une solution optimale (et garder en mémoire les valeurs des solutions optimales pour les problèmes plus petits)
- construire une solution optimale en utilisant les informations précalculées.

En pratique : après avoir obtenu une relation de récurrence liant la résolution globale à celle(s) de problèmes locaux , on initialise un tableau de taille précalculée, puis on le remplit "de bas en haut", en résolvant les problèmes locaux par taille croissante.
Exemple : calcul du $n$ème terme de la suite de Fibonacci : $F_n = F_{n-1} + F_{n-2}$

Le principe d'optimalité de Bellman permet d'obtenir une formulation récursive, puis une résolution récursive d'un problème donné.
Un algorithme de programmation dynamique procède généralement comme suit :

- caractériser ce qu'est une solution optimale
- définir (généralement par récurrence) une valeur optimale
- calculer la valeur d'une solution optimale (et garder en mémoire les valeurs des solutions optimales pour les problèmes plus petits)
- construire une solution optimale en utilisant les informations précalculées.

En pratique : après avoir obtenu une relation de récurrence liant la résolution globale à celle(s) de problèmes locaux , on initialise un tableau de taille précalculée, puis on le remplit "de bas en haut", en résolvant les problèmes locaux par taille croissante.
Exemple : calcul du $n$ème terme de la suite de Fibonacci : $F_n = F_{n-1} + F_{n-2}$ On garde en mémoire les solutions des sous-problèmes servant à construire la solution d'un problème donné.

Le principe d'optimalité de Bellman permet d'obtenir une formulation récursive, puis une résolution récursive d'un problème donné.
Un algorithme de programmation dynamique procède généralement comme suit :

- caractériser ce qu'est une solution optimale
- définir (généralement par récurrence) une valeur optimale
- calculer la valeur d'une solution optimale (et garder en mémoire les valeurs des solutions optimales pour les problèmes plus petits)
- construire une solution optimale en utilisant les informations précalculées.

En pratique : après avoir obtenu une relation de récurrence liant la résolution globale à celle(s) de problèmes locaux , on initialise un tableau de taille précalculée, puis on le remplit "de bas en haut", en résolvant les problèmes locaux par taille croissante.
Exemple : calcul du $n$ème terme de la suite de Fibonacci : $F_n = F_{n-1} + F_{n-2}$ On garde en mémoire les solutions des sous-problèmes servant à construire la solution d'un problème donné. La programmation dynamique procède de manière ascendante : on calcule d'abord les solutions des plus petits problèmes, pour construire des solutions à des problèmes plus grands, etc. Lorsqu'on utilise une approche descendante, on parle de **mémoïsation** : dès qu'un calcul est effectué, son résultat est mis en mémoire.

Le principe d'optimalité de Bellman permet d'obtenir une formulation récursive, puis une résolution récursive d'un problème donné.
Un algorithme de programmation dynamique procède généralement comme suit :

- caractériser ce qu'est une solution optimale
- définir (généralement par récurrence) une valeur optimale
- calculer la valeur d'une solution optimale (et garder en mémoire les valeurs des solutions optimales pour les problèmes plus petits)
- construire une solution optimale en utilisant les informations précalculées.

En pratique : après avoir obtenu une relation de récurrence liant la résolution globale à celle(s) de problèmes locaux , on initialise un tableau de taille précalculée, puis on le remplit "de bas en haut", en résolvant les problèmes locaux par taille croissante.
Exemple : calcul du $n$ème terme de la suite de Fibonacci : $F_n = F_{n-1} + F_{n-2}$ On garde en mémoire les solutions des sous-problèmes servant à construire la solution d'un problème donné. La programmation dynamique procède de manière ascendante : on calcule d'abord les solutions des plus petits problèmes, pour construire des solutions à des problèmes plus grands, etc. Lorsqu'on utilise une approche descendante, on parle de **mémoïsation** : dès qu'un calcul est effectué, son résultat est mis en mémoire.

Définition

Un problème possède la propriété de sous-structure optimale si une solution optimale contient la solution optimale des sous-problèmes.

Exercices

Exercice : Écrire une fonction qui calcule le *n*ème terme de la suite de Fibonacci, de façon naïve, par programmation dynamique, par mémoïsation, en OCaml et en C.

La **distance de Levenshtein** entre deux mots sur un alphabet est la distance entre ces mots comme sommets d'un graphe, dont les arêtes relient deux mots qui n'ont qu'un caractère différent en une place donné (**substitution**), ou deux mots dont l'un s'obtient de l'autre par suppression d'un caractère (**insertion** ou **suppression**).

Exercices

Exercice : Écrire une fonction qui calcule le *n*ème terme de la suite de Fibonacci, de façon naïve, par programmation dynamique, par mémoïsation, en OCaml et en C.

La **distance de Levenshtein** entre deux mots sur un alphabet est la distance entre ces mots comme sommets d'un graphe, dont les arêtes relient deux mots qui n'ont qu'un caractère différent en une place donné (**substitution**), ou deux mots dont l'un s'obtient de l'autre par suppression d'un caractère (**insertion** ou **suppression**).
Soient u et v deux mots de taille respectives n et m. Le calcul de la distance d'édition de Levenshtein entre u et v peut se faire par programmation dynamique, en remplissant un tableau des distances entre préfixes de deux mots :

Exercices

Exercice : Écrire une fonction qui calcule le *n*ème terme de la suite de Fibonacci, de façon naïve, par programmation dynamique, par mémoïsation, en OCaml et en C.

La **distance de Levenshtein** entre deux mots sur un alphabet est la distance entre ces mots comme sommets d'un graphe, dont les arêtes relient deux mots qui n'ont qu'un caractère différent en une place donné (**substitution**), ou deux mots dont l'un s'obtient de l'autre par suppression d'un caractère (**insertion** ou **suppression**).
Soient u et v deux mots de taille respectives n et m. Le calcul de la distance d'édition de Levenshtein entre u et v peut se faire par programmation dynamique, en remplissant un tableau des distances entre préfixes de deux mots :

- la distance d'un mot u à un mot vide est la taille de u.
- la distance $d(u\alpha, v\beta)$ d'un mot $u\alpha$ à un mot $v\beta$ (α, β des lettres) est le minimum parmi $d(u\alpha, v)+1$, $d(u, v\beta)+1$ et $d(u, v)+\delta_{\alpha\neq\beta}$

Exercices

Exercice : Écrire une fonction qui calcule le $n$ème terme de la suite de Fibonacci, de façon naïve, par programmation dynamique, par mémoïsation, en OCaml et en C.

La **distance de Levenshtein** entre deux mots sur un alphabet est la distance entre ces mots comme sommets d'un graphe, dont les arêtes relient deux mots qui n'ont qu'un caractère différent en une place donné (**substitution**), ou deux mots dont l'un s'obtient de l'autre par suppression d'un caractère (**insertion** ou **suppression**).
Soient u et v deux mots de taille respectives n et m. Le calcul de la distance d'édition de Levenshtein entre u et v peut se faire par programmation dynamique, en remplissant un tableau des distances entre préfixes de deux mots :

- la distance d'un mot u à un mot vide est la taille de u.
- la distance $d(u\alpha, v\beta)$ d'un mot $u\alpha$ à un mot $v\beta$ (α, β des lettres) est le minimum parmi $d(u\alpha, v)+1$, $d(u, v\beta)+1$ et $d(u, v)+\delta_{\alpha\neq\beta}$

Complexité (temporelle et spatiale) : $n \times m$.
Exercice : coder en caml et en c.

Table of Contents

Algorithme glouton

Certains problèmes d'optimisation sont tels qu'on peut espérer les résoudre par une suite de choix locaux optimums. Cette approche est généralement peu coûteuse.

Algorithme glouton

Certains problèmes d'optimisation sont tels qu'on peut espérer les résoudre par une suite de choix locaux optimums. Cette approche est généralement peu coûteuse.

Définition

Un algorithme est dit **glouton** (*greedy algorithm*) lorsqu'il progresse par amélioration de solutions partielles vers une solution finale, qu'on espère optimal, par choix locaux optimums.

Algorithme glouton

Certains problèmes d'optimisation sont tels qu'on peut espérer les résoudre par une suite de choix locaux optimums. Cette approche est généralement peu coûteuse.

Définition

Un algorithme est dit **glouton** (*greedy algorithm*) lorsqu'il progresse par amélioration de solutions partielles vers une solution finale, qu'on espère optimal, par choix locaux optimums.
Il est dit **exact** lorsque la solution qu'il fournit est optimale, et **heuristique gloutonne** lorsqu'il ne fournit pas systématiquement la meilleure solution.

Exemple : rendu de monnaie

On dispose d'un système monétaire, c'est-à-dire d'une liste de valeurs distinctes, et on souhaite représenter un prix, avec un minimum d'éléments de ce système.

Exemple : rendu de monnaie

On dispose d'un système monétaire, c'est-à-dire d'une liste de valeurs distinctes, et on souhaite représenter un prix, avec un minimum d'éléments de ce système. C'est le problème du rendu de monnaie.

Exemple : rendu de monnaie

On dispose d'un système monétaire, c'est-à-dire d'une liste de valeurs distinctes, et on souhaite représenter un prix, avec un minimum d'éléments de ce système. C'est le problème du rendu de monnaie. L'algorithme suivant est un algorithme glouton :

```
tant que (somme != 0) faire :
     piece = choix plus grande valeur
     payer piece
     somme = somme-piece
```

A chaque étape, on fait le choix qui est localement optimal. Dans certains systèmes monétaires, cet algorithme fournit toujours une solution optimale.

Exemple : rendu de monnaie

On dispose d'un système monétaire, c'est-à-dire d'une liste de valeurs distinctes, et on souhaite représenter un prix, avec un minimum d'éléments de ce système. C'est le problème du rendu de monnaie. L'algorithme suivant est un algorithme glouton :

```
tant que (somme != 0) faire :
     piece = choix plus grande valeur
     payer piece
     somme = somme-piece
```

A chaque étape, on fait le choix qui est localement optimal. Dans certains systèmes monétaires, cet algorithme fournit toujours une solution optimale.
Dans le système $(2, 6, 8)$, comment décomposer 12 ? L'algorithme glouton renvoie $8, 2, 2$, alors que la solution $6, 6$ est meilleure.

Exemple : rendu de monnaie

On dispose d'un système monétaire, c'est-à-dire d'une liste de valeurs distinctes, et on souhaite représenter un prix, avec un minimum d'éléments de ce système. C'est le problème du rendu de monnaie. L'algorithme suivant est un algorithme glouton :

```
tant que (somme != 0) faire :
     piece = choix plus grande valeur
     payer piece
     somme = somme-piece
```

A chaque étape, on fait le choix qui est localement optimal. Dans certains systèmes monétaires, cet algorithme fournit toujours une solution optimale.
Dans le système $(2, 6, 8)$, comment décomposer 12 ? L'algorithme glouton renvoie $8, 2, 2$, alors que la solution $6, 6$ est meilleure.
Dans le système $(2, 5, 8)$, comment décomposer 9 ? L'algorithme glouton ne trouve pas de solution, alors qu'il en existe une : $5, 2, 2$.

Exemple : parcours le plus court

On souhaite, partant de $(0,0)$, passer par tous les points d'un ensemble, en minimisant la distance parcourue.

```
tant que (il reste un point) faire :
     arret = point le plus proche
     retirer arret de la liste des points
     aller à arret
```

Par cet algorithme, tous les points seront visités.

Exemple : parcours le plus court

On souhaite, partant de $(0,0)$, passer par tous les points d'un ensemble, en minimisant la distance parcourue.

```
tant que (il reste un point) faire :
     arret = point le plus proche
     retirer arret de la liste des points
     aller à arret
```

Par cet algorithme, tous les points seront visités. Exemple de non-optimalité ?

Exemple : parcours le plus court

On souhaite, partant de $(0,0)$, passer par tous les points d'un ensemble, en minimisant la distance parcourue.

```
tant que (il reste un point) faire :
     arret = point le plus proche
     retirer arret de la liste des points
     aller à arret
```

Par cet algorithme, tous les points seront visités. Exemple de non-optimalité ? En partant de $O = (0,0)$:
$\{(0,1),(0,2),....(0,13),(1.1,0)\}$

Table of Contents

Codage de Huffman

La construction d'un code de Huffman se fait par un algorithme glouton : on sélectionne une solution locale optimale pour améliorer une solution locale, jusqu'à une solution globale.

Codage de Huffman

La construction d'un code de Huffman se fait par un algorithme glouton : on sélectionne une solution locale optimale pour améliorer une solution locale, jusqu'à une solution globale. On appelle optimal ici tout arbre T construit sur l'alphabet A minimisant la quantité $Q_A(T) = \sum_{a \in A} p(a)l(a)$, où $p(a)$ est la probabilité associée à la lettre a, et $l(a)$ sa longueur de code (qui est aussi sa profondeur dans l'arbre T).

1. Un arbre créé par cet algorithme est binaire strict ;

Codage de Huffman

La construction d'un code de Huffman se fait par un algorithme glouton : on sélectionne une solution locale optimale pour améliorer une solution locale, jusqu'à une solution globale. On appelle optimal ici tout arbre T construit sur l'alphabet A minimisant la quantité $Q_A(T) = \sum_{a \in A} p(a)l(a)$, où $p(a)$ est la probabilité associée à la lettre a, et $l(a)$ sa longueur de code (qui est aussi sa profondeur dans l'arbre T).

1. Un arbre créé par cet algorithme est binaire strict ;
2. Soient x et y deux lettres de poids minimal ; il existe un arbre créé par cet algorithme qui est optimal dans lequel ces lettres sont soeurs et à profondeur maximale ;

Codage de Huffman

La construction d'un code de Huffman se fait par un algorithme glouton : on sélectionne une solution locale optimale pour améliorer une solution locale, jusqu'à une solution globale. On appelle optimal ici tout arbre T construit sur l'alphabet A minimisant la quantité $Q_A(T) = \sum_{a \in A} p(a)l(a)$, où $p(a)$ est la probabilité associée à la lettre a, et $l(a)$ sa longueur de code (qui est aussi sa profondeur dans l'arbre T).

1. Un arbre créé par cet algorithme est binaire strict ;
2. Soient x et y deux lettres de poids minimal ; il existe un arbre créé par cet algorithme qui est optimal dans lequel ces lettres sont soeurs et à profondeur maximale ;
3. (**Récurrence**)P_n : pour tout alphabet de taille n, l'algorithme fournit un arbre optimal.

Codage de Huffman

La construction d'un code de Huffman se fait par un algorithme glouton : on sélectionne une solution locale optimale pour améliorer une solution locale, jusqu'à une solution globale. On appelle optimal ici tout arbre T construit sur l'alphabet A minimisant la quantité $Q_A(T) = \sum_{a \in A} p(a)l(a)$, où $p(a)$ est la probabilité associée à la lettre a, et $l(a)$ sa longueur de code (qui est aussi sa profondeur dans l'arbre T).

1. Un arbre créé par cet algorithme est binaire strict ;
2. Soient x et y deux lettres de poids minimal ; il existe un arbre créé par cet algorithme qui est optimal dans lequel ces lettres sont soeurs et à profondeur maximale ;
3. (**Récurrence**)P_n : pour tout alphabet de taille n, l'algorithme fournit un arbre optimal.
   - Pour un alphabet de taille 1 ou 2, l'algorithme fournit évidemment un arbre optimal.

Codage de Huffman

La construction d'un code de Huffman se fait par un algorithme glouton : on sélectionne une solution locale optimale pour améliorer une solution locale, jusqu'à une solution globale. On appelle optimal ici tout arbre T construit sur l'alphabet A minimisant la quantité $Q_A(T) = \sum_{a \in A} p(a)l(a)$, où $p(a)$ est la probabilité associée à la lettre a, et $l(a)$ sa longueur de code (qui est aussi sa profondeur dans l'arbre T).

1. Un arbre créé par cet algorithme est binaire strict ;
2. Soient x et y deux lettres de poids minimal ; il existe un arbre créé par cet algorithme qui est optimal dans lequel ces lettres sont soeurs et à profondeur maximale ;
3. (**Récurrence**)P_n : pour tout alphabet de taille n, l'algorithme fournit un arbre optimal.
   - Pour un alphabet de taille 1 ou 2, l'algorithme fournit évidemment un arbre optimal.
   - Soient x et y de poids minimal dans cet alphabet A de taille $n+1$. Construisons l'alphabet B en enlevant les lettres x et y et en les remplaçant par la lettre z, de poids $p(z) = p(x) + p(y)$. **Par HR**, l'algorithme construit un arbre optimal T' sur l'alphabet B, lequel est un certain T sur A
   - Soit U sur l'alphabet A, optimal, avec x et y feuilles soeurs de profondeur maximale, et U' l'arbre qu'on en déduit sur B. Par optimalité de T', $Q_B(T') \leq Q_B(U')$, on en déduit que $Q_B(T) \leq Q_B(U)$, et donc que T est optimal sur A.

Sélection d ’activité

On dispose d’un ensemble d’activités, dont on connait l’heure de début et l’heure de fin. Comment choisir un ensemble maximal d’activités compatibles ?

Sélection d 'activité

On dispose d'un ensemble d'activités, dont on connait l'heure de début et l'heure de fin. Comment choisir un ensemble maximal d'activités compatibles ?

- Trier par début croissant ; sélectionner le cours qui débute le plus tôt
- Trier par durée croissante ; sélectionner le cours le plus cours
- Trier par heure de fin croissante ; sélectionner le cours qui finit le plus tôt

Sélection d 'activité

On dispose d'un ensemble d'activités, dont on connait l'heure de début et l'heure de fin. Comment choisir un ensemble maximal d'activités compatibles ?

- Trier par début croissant ; sélectionner le cours qui débute le plus tôt
- Trier par durée croissante ; sélectionner le cours le plus cours
- Trier par heure de fin croissante ; sélectionner le cours qui finit le plus tôt
- sélectionner le cours qui crée le moins d'incompatibilité ; celui qui commence le plus tard, etc...

Sélection d'activité

Complexité de l'algorithme choisi ?

Sélection d'activité

Complexité de l'algorithme choisi ? Le tri est en $O(n \log n)$.

Sélection d'activité

Complexité de l'algorithme choisi ? Le tri est en $O(n \log n)$.

Proposition

L'algorithme glouton qui choisit le cours qui finit le plus tôt est optimal

Sélection d'activité

Complexité de l'algorithme choisi ? Le tri est en $O(n \log n)$.

Proposition

L'algorithme glouton qui choisit le cours qui finit le plus tôt est optimal

Preuve : On suppose l'ensemble $(C_i)_i$ des cours triés par heure de fin croissante.
Il existe une solution optimale contenant C_1. En effet, si ce n'était pas le cas dans une solution optimale donnée, le premier des cours sélectionnés finirait après C_1 et pourrait être remplacé par celui-ci.
Il existe une solution optimale contenant C_1 et une solution optimale sur l'ensemble E des cours compatibles avec C_1 : soit une solution $(C_1, D_2, \dots D_k)$ optimale, qui existe d'après ce qui précède. Les D_i sont compatibles avec C_1. S'ils ne formaient pas une solution optimale pour E, il en existe une meilleure, strictement plus grande, dont tous les cours sont compatibles avec C_1 ce qui est absurde.
Il existe donc une solution optimale qui sera construite par l'algorithme glouton.
La preuve peut ensuite de faire par récurrence : $\mathscr{P}$ est la propriété : "Soient les k premiers choix de l'algorithme, il existe une solution optimale comportant ces k premiers choix."

Ordonnancement de tâches

On cherche à établir un ordre pour planifier des taches unitaires avec pénalité de retard sur une machine unique. Chaque tâche T_i possède une durée unitaire et une date de fin d_i, à laquelle on souhaite qu'elle soit achevée, sans quoi, il faut payer une pénalité v_i. Comment réaliser ces tâches en minimisant le cumul des pénalités ?

Ordonnancement de tâches

On cherche à établir un ordre pour planifier des taches unitaires avec pénalité de retard sur une machine unique. Chaque tâche T_i possède une durée unitaire et une date de fin d_i, à laquelle on souhaite qu'elle soit achevée, sans quoi, il faut payer une pénalité v_i. Comment réaliser ces tâches en minimisant le cumul des pénalités ?
Trier par pénalité décroissante, terminer les tâches au plus proche de leur date de fin prévue. Chaque tâche est traitée à son tour et planifiée pour se terminer au plus proche de sa date de fin. Si on ne peut trouver de tel créneau, on doit payer la pénalité et on place cette tâche le plus tard possible.