# Algorithmes gloutons


| | | |
|---|---|---|
| **I** | **Principe** | **1** |
| I.1 | Problème d'optimisation combinatoire | 1 |
| I.2 | Algorithme glouton | 2 |
| I.3 | Cas du rendu de monnaie | 2 |
| **II** | **Construction de l'arbre de Huffman** | **3** |
| II.1 | Description | 3 |
| II.2 | Algorithme glouton et implémentation | 4 |
| II.3 | Preuve d'optimalité | 5 |
| **III** | **Sélection d'activités** | **6** |
| III.1 | Description | 6 |
| III.2 | Algorithme glouton et implémentation | 6 |
| III.3 | Preuve d'optimalité | 7 |
| **IV** | **Principe général des preuves d'optimalité** | **8** |
| **V** | **Ordonnancement de tâches** | **9** |
| V.1 | Description | 9 |
| V.2 | Algorithme glouton et implémentation | 10 |
| V.3 | Preuve d'optimalité | 11 |
| **VI** | **Exercices** | **12** |


# I Principe

## I.1 Problème d'optimisation combinatoire

On considère ici un problème d'énumération comme dans le chapitre sur le retour sur trace : un ensemble **fini** $S$ muni d'une valeur $s_0 \in S$ appelée la **position** initiale et de deux fonctions

$$mouv : S \to \mathcal{P}(S)$$

$$taille : S \to \mathbb{N}$$

telles que $taille(s_0) = 0$ et que

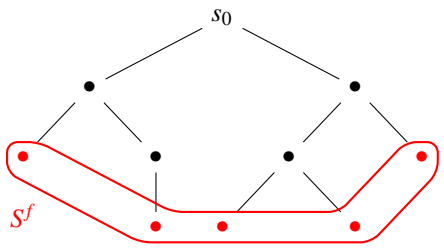
$$\forall x, y \in S, x \in mouv(y) \Rightarrow taille(x) < taille(y)$$

Cette dernière propriété assure qu'en partant de $s_0$ et en effectuant une série de *mouvements* on aboutit forcément à une position finale, c'est-à-dire pour laquelle $mouv(s) = \emptyset$. On note $S^f \subset S$ les positions finales.

On peut donc ainsi tracer l'arbre des positions où $s_0$ est à la racine et les enfants d'une position sont celles données par $mouv$. Attention, dans cet arbre, on peut distinguer deux positions selon les mouvements qui nous y amènent. En fait, si on partage ces positions, il s'agit d'un graphe acyclique non orienté.

**Exercice 1** Prouver cette affirmation.

Sur la représentation suivante, on a fait figurer en rouge les éléments de $S^f$.



Dans le retour sur trace, on cherchait une position finale vérifiant une certaine propriété. Ici, on considère une fonction objectif :

$$\chi : S^f \to \mathbb{R}$$

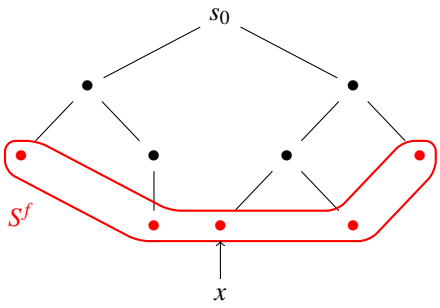
qui associe une valeur numérique à chaque position. On considère un problème dit de maximisation : déterminer $x \in S$, tel que $\chi(x) = \max_{y \in S} \chi(y)$. On notera souvent $x = argmax_{y \in S}\, \chi(y)$. Quand le problème considéré dépend d'une entrée $E$, on note $Opt(E)$ la valeur de l'optimal.

**Remarque**

- En considérant $g : y \mapsto -f(y)$, on transforme un problème de maximisation en un problème de minimisation. On parle plus généralement de problèmes d'optimisation combinatoire.
- Il y a une ambiguïté sur $\text{argmax}_{y \in S} f(y)$ quand plusieurs éléments de $S$ réalisent ce maximum. Dans la plupart des algorithmes gloutons qu'on va considérer, on commence par donner un ordre sur $S$ et on considère le plus petit $y$ pour cet ordre réalisant le maximum. L'ordre choisi est alors crucial dans la preuve de correction. C'est aussi une des raisons pour lesquelles les algorithmes gloutons sont souvent de complexité temporelle $O(n \log_2 n)$.

Une première stratégie très élémentaire consiste alors à énumérer $S$, de manière exhaustive ou avec une stratégie plus fine comme le retour sur trace, puis à déterminer un élément maximal de manière directe.

Cela revient donc à déterminer l'arbre des solutions puis à trouver une feuille maximisant l'objectif :




## I.2 Algorithme glouton

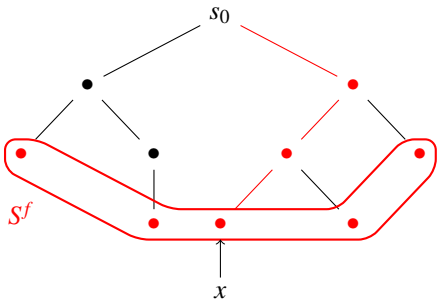
Le nombre de nœud de l'arbre est en général exponentiel en la taille de l'entrée. Ainsi, une exploration exhaustive est optimale mais très coûteuse.

Un algorithme glouton va suivre une approche beaucoup plus efficace : à chaque étape de construction de la solution, on choisit la branche qui maximise la fonction d'objectif. Pour cela, on considère que la fonction d'objectif $\chi$ se prolonge à toutes les positions dans $S$.

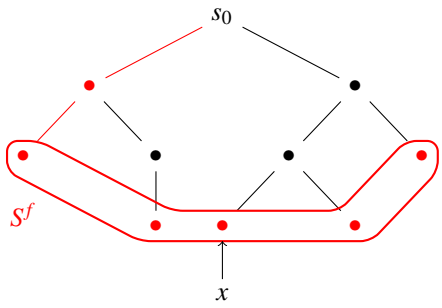
Si la position courante est $x \in S$, la position suivante sera $y = argmax_{z \in mouv(x)}\, \chi(z)$.

On dit que $y$ est le **mouvement glouton** ou **choix glouton**.

Sur l'arbre précédent, cela reviendrait à n'emprunter qu'une seule branche indiquée en rouge sur le schéma.




Cela a l'air très efficace mais il y a un problème majeur : il n'y a aucune garantie qu'on aboutisse ainsi à une solution, encore moins à une solution optimale. En effet, on aurait très bien pu faire les choix suivants :




et ne pas aboutir à une solution.

## I.3 Cas du rendu de monnaie

Considérons par exemple le problème du **rendu de monnaie** : étant donné, une liste de valeurs faciales de pièces $P = (v_1, \dots, v_p) \in (\mathbb{N}^*)^p$ avec $1 = v_1 < \cdots < v_p$ et une somme $n \in \mathbb{N}^*$, on cherche la manière d'exprimer cette somme avec le plus petit nombre de pièces possible.

Plus précisément, l'ensemble des positions est $S = \{(k_1, \dots, k_p) \in N^p \mid k_1v_1 + \cdots + k_pv_p \leq n\}$, les positions finales est $S^f = \{(k_1, \dots, k_p) \in N^p \mid k_1v_1 + \cdots + k_pv_p = n\}$ et la fonction d'objectif est $\chi : (k_1, \dots, k_p) \mapsto k_1 + \cdots + k_p$.

On cherche alors $x = argmin_{y \in S^f} \chi(y)$.

Comme $1 = v_1$, $S \neq \emptyset$ car $(n, 0, \dots, 0) \in S$ et ainsi $\chi(x) \leq n$.

**Exemple** Si $P = (1, 2, 5, 10)$ et $n = 14$ on a $(3, 3, 1, 0) \in S^f$ car $n = 3 + 3 \times 2 + 1 \times 5$.

L'algorithme glouton va utiliser la plus grande pièce possible à chaque étape : si on doit décomposer $n$, on cherche $p_i \leq n$ maximale, on décompose $n - p_i$ et on rajoute la pièce $i$ pour en déduire une décompositon de $n$. On s'arrête quand le nombre à décomposer vaut 0 qui correspond à l'unique rendu vide.

**Exemple** Si $P = (1, 2, 5, 10)$ et $n = 14$.

- On utilise la plus grande pièce possible $10 \leq 14$ puis on exprime $4 = 14 - 10$
- Ici, la plus grande pièce est 2 et on continue avec $2 = 4 - 2$
- La plus grande pièce est encore 2 et on s'arrête car $0 = 2 - 2$.
- En conclusion, on a obtenu $x = (0, 2, 0, 1)$.
- Une exploration exhuastive permet de s'assurer qu'on a effectivement obtenu une décomposition minimale. En effet, ici l'ensemble des décompositions est : { (14,0,0,0), (12,1,0,0), (8,3,0,0), (6,4,0,0), (4,5,0,0), (2,6,0,0), (0,7,0,0), (9,0,1,0), (7,1,1,0), (5,2,1,0), (3,3,1,0), (1,4,1,0), (4,0,2,0), (2,1,2,0), (0,2,2,0), (4,0,0,1), (2,1,0,1), (0,2,0,1) }.

**Exemple** On considère maintenant $P = (1, 2, 7, 10)$ et $n = 14$

- L'algorithme glouton va ici procéder comme dans l'exemple précéent et on va obtenir $x = (0, 2, 0, 1)$.
- Mais on remarque que ce n'est pas un minimum car $x' = (0, 0, 2, 0)$ convient avec $f(x') = 2 < 3 = f(x)$.

**Remarque** L'algorithme glouton n'est pas nécessairement optimal. Un système de pièces, on parle de système monétaire, telle que l'algorithme glouton soit optimal est appelé un *système canonique*. Un exercice en fin de chapitre demande de démontrer qu'un certain système est canonique.

On peut se poser la question des algorithmes pour lesquels l'algorithme glouton aboutit nécessairement à une solution optimale. Il se trouve qu'il y a un cadre général mais hors programme dont on pourra voir se dégager la nature à la lumière des preuves d'optimalité qui vont suivre.

# II Construction de l'arbre de Huffman

## II.1 Description

**Remarque** Ce paragraphe décrit l'étape cruciale du principe de compression de Huffman. Celui-ci sera présenté complètement dans le chapitre Algorithmique des textes.

On va étudier ici un principe de compression parfaite (sans perte d'information à la décompression) de données appelé l'algorithme de Huffman et qui repose sur ce principe simple : coder sur moins de bits les caractères les plus fréquents.

Par exemple si on considère le mot `abaabc`, en le codant avec un nombre de bits fixes, par exemple 2 avec le code `a=00,b=01,c=10`, on aurait besoin de 12 bits pour représenter le mot. Mais si on choisit le code suivant : `a=0,b=10,c=11`, il suffit de 9 bits. On a donc gagné 3 bits soit un facteur de compression de 75%.

On remarque que pour pouvoir décompreser, il n'aurait pas été possible de faire commencer le code de `b` ou `c` par un `0`, sinon on aurait eu une ambiguité avec la lecture d'un `a`. On parle alors de code préfixe :

**Définition II.1** Soit $X \subset \{0,1\}^*$, on dit que $X$ est un code préfixe lorsque pour tous $x, y \in X$, $x$ n'est pas un préfixe de $y$ et $y$ n'est pas un préfixe de $x$.

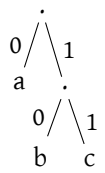
On se pose alors la question du code préfixe optimal pour un texte donné.

Plus précisément, étant donné un alphabet fini $\Sigma$ et une application $f : \Sigma \to [0,1]$ associant à chaque lettre son nombre d'occurences dans le texte considéré. Ainsi $\sum_{x\in\Sigma} f(x)$ est la longueur du texte. On cherche un code préfixe $X$ et une application $c : \Sigma \to X$ telle que $\sum_{x\in\Sigma} f(x)|c(x)|$ soit minimale car cela correspond au nombre de bits après codage.

**Remarque** On utilise aussi la notion de fréquence du lettre qui est son nombre d'occurence rapporté à la longueur du texte. Un des avantages de la notion de fréquence est qu'il est possible de considèrer une table de fréquence déjà construite comme celle de la langue française.

L'application de codage $c$ peut être représenté par un arbre binaire où les arêtes gauches correspondent à 0, les arêtes droites à 1 et les feuilles aux éléments de $\Sigma$ dont les étiquettes des chemins y menant depuis la racine de l'arbre correspondent à leur image par $c$.

Par exemple, pour le code $a = 0, b = 10, c = 11$ on aurait l'arbre :




Avec un tel arbre, il est très simple de décoder le texte codé car il suffit de suivre un chemin dans l'arbre jusqu'à tomber sur une feuille, produire la lettre correspondante, puis repartir de la racine de l'arbre. La longueur du code associé à une lettre est alors égale à la profondeur de la feuille correspondante. L'optimalité du codage préfixe est ainsi équivalente à la minimalité de l'arbre vis-à-vis de la fonction d'objectif $\varphi(t) = \sum_{x\in\Sigma} f(x)p(t,x)$ où $p(t,x)$ est la profondeur de la feuille d'étiquette $x$ dans l'arbre $t$ ou 0 si $x$ n'est pas une des étiquettes, cet extension permettant d'étendre la fonction d'objectif aux solutions partielles.

## II.2 Algorithme glouton et implémentation

L'algorithme d'Huffman va construire un arbre correspondant à un codage optimal à l'aide d'une file de priorité d'arbres. On étend pour cela l'application $f$ à de tels arbres en définissant que si $t$ est un arbre de feuilles $x_1, \ldots, x_n$ alors $f(t) = f(x_1) + \cdots + f(x_n)$.

- Au départ, on place dans la file des arbres réduits à une feuille pour chaque élément $x \in \Sigma$ et dont la priorité est $f(x)$.
- Tant que la file contient au moins deux éléments
  - ⋆ on retire les deux plus petits éléments $x$ et $y$ de la file de priorité $f(x)$ et $f(y)$
  - ⋆ on ajoute un arbre $z = Noeud(x, y)$ de priorité $f(z) = f(x) + f(y)$.
- On renvoie l'unique élément restant dans la file.

L'implémentation de cet algorithme est alors assez directe avec une file de priorité. On réutilise ici la structure de tas implementée en FIXME. Comme il s'agit d'un tas max, on insère avec $-f(x)$ comme valeur.

OCaml
```
let construit_arbre occ =
    let arbres = Tas.cree 2000 (0, Feuille 0) in
    for i = 0 to 255 do
        let f = occ.(i) in
        if f > 0 (* on ignore les occurrences nulles *)
        then Tas.insere (-f, Feuille i) arbres
    done;
    while Tas.taille arbres > 1 do
        let fx, x = Tas.supprime_racine arbres in
        let fy, y = Tas.supprime_racine arbres in
        Tas.insere (fx+fy, Noeud(x,y)) arbres
    done;
    snd (Tas.supprime_racine arbres)
```

C
```
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

L'algorithme de Huffman est un algorithme glouton car si on considère pour solution partielle la fôret présente dans la file et pour objectif la fonction $\varphi$ étendue aux fôrets en sommant la valeur de $\varphi$ sur chaque arbre, alors fusionner dans la fôret $F$ deux arbres $x$ et $y$ en la transformant en une fôret $F'$ va avoir l'impact suivant sur la fonction d'objectif :

$$\varphi(F') = \varphi(F) + f(x) + f(y)$$

car, en effet, on va rajouter 1 à la profondeur de chaque feuille et donc on passe pour la contribution de $x$ de $\varphi(x) = \sum_{c\in\Sigma} f(c)p(x,c)$ à $\sum_{c\in\Sigma} f(c)(p(x,c)+1) = \varphi(x) + \sum_{c\in\Sigma} f(c) = \varphi(x) + f(x)$.

On remarque ainsi que la fusion qui minimise localement $\varphi$ est celle qui fusionne les deux arbres de plus petite valeur pour $f$.

## II.3 Preuve d'optimalité

Pour montrer que l'algorithme glouton produit ici un codage minimal, on va utiliser une technique classique qui consiste à montrer qu'étant donné une solution optimale, on peut toujours la transformer sans augmenter sa valeur pour obtenir, de proche en proche, la solution renvoyée par le glouton.

**Théorème II.1** Supposons que les lettres les moins fréquentes soient $a$ et $b$, il existe un arbre optimal dont les deux feuilles étiquettées par $a$ et $b$ descendent du même noeud et sont de profondeur maximale.

■ **Preuve**

Considérons un arbre optimal $t$ et soient $c$ l'étiquette d'une feuille de profondeur maximale. On remarque qu'elle a forcément une feuille sœur car sinon, on pourrait omettre le noeud et l'arbre obtenu serait de plus petite valeur par $\varphi$.

FIXME : dessin

Soit $d$ l'étiquette de cette feuille sœur. Sans perte de généralités, on suppose que $f(c) \leq f(d)$ et $f(a) \leq f(b)$. Comme $a$ a le plus petit nombre d'occurences, a $f(a) \leq f(c)$ et comme $b$ est la deuxième, on a $f(b) \leq f(d)$. De plus, $p(t,a) \geq p(t,c)$ et $p(t,b) \geq p(t,d)$.

Si on échange les étiquettes $a$ et $c$, seule les termes associées à ces lettres changent dans l'évaluation de $\varphi$. Si on note $t'$ le nouvel arbre obtenu après cet échange, on a

$$\varphi(t') = \varphi(t) - f(a)p(t,a) - f(c)p(t,c) + f(a)p(t,c) + f(c)p(t,a)$$

Or, $f(c) \geq f(a)$ et $p(t,a) \geq p(t,c)$ donc

$$\varphi(t') = \varphi(t) + (f(c) - f(a))(p(t,a) - p(t,c)) \leq \varphi(t)$$

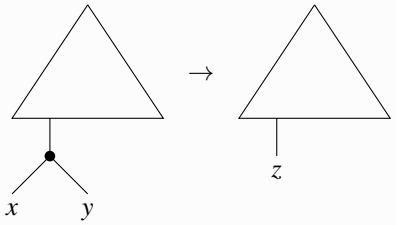
L'échange préserve le caractère optimal. En fait, ici, on a nécessairement une égalité pour ne pas aboutir à une contradiction, donc soit les feuilles étaient à même profondeur, soit les lettres avaient le même nombre d'occurrences.

Comme on a les mêmes relations entre $b$ et $d$, on peut effectuer le même argument et échanger les étiquettes en préservant le caractère optimal.

■

Le théorème suivant permet de raisonner par récurrence en diminuant le nombre de lettres.

**Théorème II.2** Soit $t$ un arbre ayant $x$ et $y$ comme feuilles soeurs et $t'$ l'arbre obtenu en remplaçant le noeud liant $x$ et $y$ par une feuille étiquettée par $z$ où $z$ est une nouvelle lettre telle que $f(z) = f(x) + f(y)$.




On a alors $\varphi(t) = \varphi(t') + f(z)$.

■ **Preuve**

Seule les termes portant sur $x, y$ et $z$ sont influencés par le changement et on a :

$$\begin{aligned}\varphi(t) &= \varphi(t') + f(x)p(t,x) + f(y)p(t,y) - f(z)p(t',z)\\ &= \varphi(t') + f(z)(p(t',z)+1) - f(z)p(t',z)\\ &= \varphi(t') + f(z)\end{aligned}$$

■

**Théorème II.3** L'algorithme de Huffman renvoie un arbre optimal.

■ **Preuve**

Par récurrence sur $|\Sigma|$.

*Initialisation* : si $\Sigma$ ne contient qu'une lettre, il n'y a qu'un arbre qui est nécessairement optimal.

*Hérédité* : si la propriété est vraie pour un alphabet de $n-1 \geq 1$ lettres, alors soit $\Sigma$ contenant $n$ lettres et $x$ et $y$ les deux lettres les moins fréquentes.

On pose $\Sigma'$ obtenue en remplaçant $x$ et $y$ par une nouvelle lettre $z$ et on suppose que $f(z) = f(x) + f(y)$. L'hypothèse de récurrence assure qu'on obtient un arbre optimal $t'$ en appliquant l'algorithme d'Huffman sur $\Sigma'$. Comme la première étape d'Huffman va fusionner les feuilles $x$ et $y$, on sait que l'arbre $t$ obtenu en partant de $\Sigma$ se déduit de $t'$ en remplaçant $z$ par $Noeud(x,y)$. Le théorème précédent assure alors que $\varphi(t) = \varphi(t') + f(z)$.

Soit $t_o$ un arbre optimal pour $\Sigma$ dans lequel $x$ et $y$ sont soeurs, possible en vertu du premier théorème, et soit $t'_o$ l'arbre obtenue en remplaçant dans $t_o$ le noeud liant $x$ et $y$ par une feuille étiquettée par $z$. On a ici encore $\varphi(t_o) = \varphi(t'_o) + f(z) \geq \varphi(t') + f(z) \geq \varphi(t)$ car $t'$ est optimal.

Ainsi, on a bien l'égalité $\varphi(t_o) = \varphi(t)$ et $t$ est optimal.

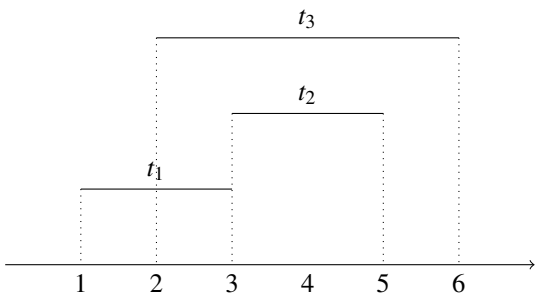
■

# III Sélection d'activités

## III.1 Description

Étant donné un ensemble d'activités données par leur temps de début et leur temps de fin (on considère les temps comme des entiers pour simplifier), on se pose la question du nombre maximal d'activité que l'on puisse sélectionner sans que deux activités soient en conflits. Cela correspond par exemple à l'organisation du planning d'un employé.

On dit que deux activités $(d_1, f_1)$ et $(d_2, f_2)$ sont en conflits quand $[d_1, f_1[ \cap [d_2, f_2[ \neq \emptyset$.




Ici, $t_1$ et $t_2$ sont en conflits avec $t_3$. Mais $t_1$ et $t_2$ ne sont pas en conflit. On considère que deux activités peuvent se succéder directement : $f_1 = d_2$.

On considère donc en entrée de ce problème une suite finie $((d_1, f_1), \ldots, (d_n, f_n))$ et on cherche un sous-ensemble $I \subset [\![1, n]\!]$ de plus grand cardinal tel que pour tous $i, j \in I$, si $i \neq j$ alors $(d_i, f_i)$ et $(d_j, f_j)$ ne sont pas en conflits. On dit que $I$ est un **ensemble indépendant**.

## III.2 Algorithme glouton et implémentation

Pour résoudre ce problème, on considère l'algorithme glouton associé à la fonction d'objectif cardinal et **en triant les activités** ordre croissant de temps de fin.

Cet algorithme est implémenté dans le programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    unsigned int id;
    unsigned int debut;
    unsigned int fin;
    unsigned char selectionnee;
} activite;

int compare_activites(const void *t1, const void *t2)
{
    return ((activite *)t1)->fin - ((activite *)t2)->fin;
}

void selectionne(activite *activites, size_t nb_activites)
{
    size_t derniere_activite = 0;
    /* on commmence par trier en O(n log2 n) les activites
     * selon le temps de fin */
    qsort(activites, nb_activites,
        sizeof(activite), compare_activites);
    activites[0].selectionnee = 1;
    for (size_t i = 1; i < nb_activites; i++)
    {
        if (activites[i].debut >= activites[derniere_activite].fin)
        {
            activites[i].selectionnee = 1;
            derniere_activite = i;
        }
    }
}

int main()
{
    activite activites[] = {
        { 0, 1, 3, 0 }, { 1, 3, 4, 0 }, { 2, 2, 5, 0 },
        { 3, 5, 9, 0 }, { 4, 11, 12, 0 }, { 5, 8, 10, 0 },
        { 6, 0, 7, 0 }
    };

    size_t nb_activites = sizeof(activites) / sizeof(activite);

    selectionne(activites, nb_activites);

    for (size_t i = 0; i < nb_activites; i++)
    {
        printf("Activité %d (%d,%d) : %d\n",
                activites[i].id, activites[i].debut,
                activites[i].fin, activites[i].selectionnee);
    }

    return 0;
}
```

Ce programme produit alors la sortie :

```
Activité 0 (1,3)  : 1
Activité 1 (3,4)  : 1
Activité 2 (2,5)  : 0
Activité 6 (0,7)  : 0
Activité 3 (5,9)  : 1
Activité 5 (8,10)  : 0
Activité 4 (11,12)  : 1
```

**Remarque** Comme l'algorithme commence par effectuer un tri, on a rajouté dans la structure `activite` un champ permettant d'identifier une activité autrement que par son indice.

## III.3 Preuve d'optimalité

On va prouver que l'algorithme glouton renvoie un ensemble indépendant optimal. Le fait que l'ensemble soit indépendant étant direct, on se concentre sur la preuve d'optimalité en présentant un schéma de preuve qui correspond à celui identifié dans le paragraphe précédent.

**Théorème III.1** Si $a_1, \dots, a_n$ sont des activités énumérées dans l'ordre croissant de leur temps de fin, alors il existe un ensemble indépendant optimal contenant $a_1$.

**Remarque** Cela signifie qu'il fait le même choix que l'algorithme glouton à la première étape.

**Preuve**

Soit $I$ un ensemble indépendant optimal ne contenant pas $a_1 = (d_1, f_1)$ (sinon c'est direct). Si $a_k = (d_k, f_k)$ est l'activité de plus petit indice dans $I$, alors $f_k \geq f_1$ donc pour tout $a_i = (d_i, f_i)$ dans $I' = I \backslash \{a_k\}$ on a $d_i \geq f_k \geq f_1$ et ainsi $a_1$ et $a_i$ ne sont pas en conflit. Ainsi $I' \cup \{a_1\}$ est un ensemble indépendant contenant $a_1$ de même cardinal que $I$ donc optimal.

■

**Théorème III.2** Soit $A = \{a_1, \dots, a_n\}$ des activités ordonnées par ordre croissant de temps de fin et $I$ un ensemble indépendant optimal contenant $a_1 = (d_1, f_1)$ (ce qui est possible selon le théorème précédent).
$I' = I \backslash \{a_1\}$ est optimal pour $A' = \{\ (d, f) \in A \mid d \geq f_1\ \}$.

**Preuve**

Si, par l'absurde, $I'$ est pas optimal pour $A'$ alors $J \subset A'$ est un ensemble indépendant de cardinal strictement plus grand que celui de $I'$. Or, $A' \cup \{a_1\}$ est indépendant pour l'ensemble des activités et est de cardinal strictement plus grand que $I$. Contradiction.

■

**Théorème III.3** L'algorithme glouton renvoie un ensemble indépendant optimal.

**Preuve**

Par récurrence forte sur le nombre d'activités.

- Initialisation : Pour une activité $a_1$, le glouton renvoie $\{a_1\}$ qui est directement optimal.
- Hérédité : Si la propriété est vérifiée pour $k \leq n$ activités, soit $A = \{a_1, \dots, a_n\}$ des activités ordonnées par temps de fin. Soit $I$ un ensemble indépendant optimal contenant $a_1$ et $I' = I \backslash \{a_1\}$. Le théorème précédent assure que $I'$ est optimal sur $A' = \{\ (d, f) \in \mid d \geq f_1\ \}$.
  Par hypothèse de récurrence, l'algorithme glouton sur $A'$ produit un ensemble indépendant optimal $G'$, donc tel que $|G'| = |I'|$. Par construction l'algorithme glouton sur $A$ renvoie $G = G' \cup \{a_1\}$ de même cardinal que $I$, donc optimal.

■

# IV Principe général des preuves d'optimalité

L'idée de la preuve précédente et de celle qui vont suivre est d'identifier dans la solution gloutonne ce qui correspond au choix glouton initial. On va reprendre de manière générale le principe de cette preuve ici dans le cas d'un problème de minimisation.

Si on note $g$ le glouton pour l'entrée $E$ et $c$ ce choix, on peut déduire de $E$ la sous-entrée de $E'$ correspondant à avoir enlevé $c$, ce qui peut conduire à changer $E$ ou à supprimer d'autres éléments conséquences du choix $c$. Comme le glouton est une itération de choix gloutons, on déduit $g'$ de $g$ en omettant ce choix $c$ et $g'$ est le résultat du glouton pour $E'$.

*Pour Huffman, $E'$ se déduit en rajoutant une nouvelle lettre, ce n'est donc pas juste $E$ sans les deux lettres les moins fréquentes.*

On va démontrer un premier résultat qui est un résultat d'échange : si le glouton fait le choix $c$ c'est que $c$ est localement minimal, on va alors montrer qu'il existe un optimal $o'$ faisant le choix $c$ en montrant que si un optimal $o$ a fait un choix minimal $c'$ alors on peut préserver l'optimalité en remplaçant $c'$ par $c$.

*Dans le cas de Huffman, il s'agissait de placer les deux lettres les moins fréquentes en tant que sœurs d'un nœud profond.*

Ensuite, on montre un résultat de réduction en enlevant $c$ qui va relier $\chi(g)$ à $\chi(g')$ et, de même, $\chi(o')$ à $\chi(o'')$, car on a effectué la même omission de $c$ pour le déduire. On s'assure que cette relation permette alors de faire le raisonnement suivant :

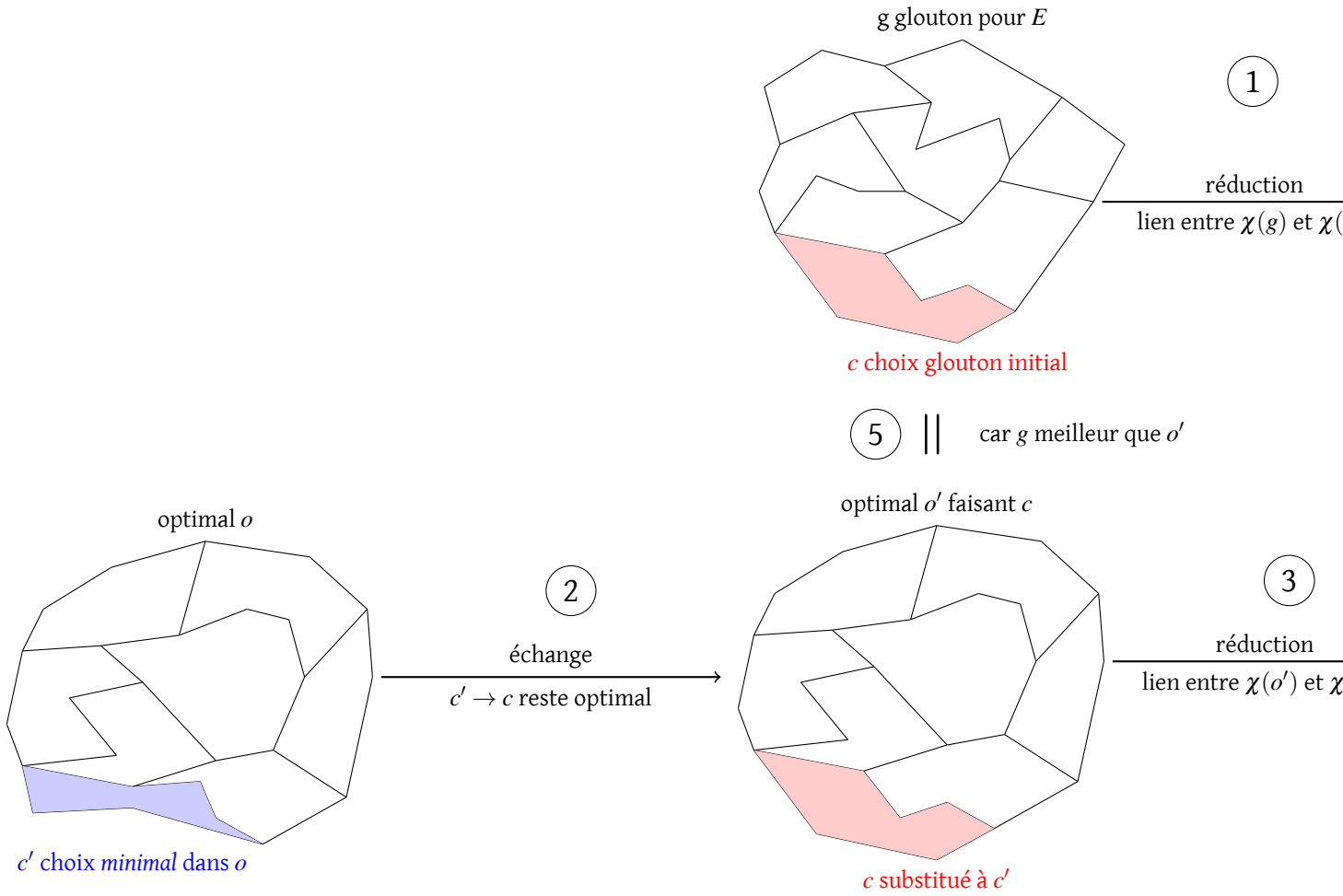
$$\chi(g) = \text{expr. de } \chi(g') \text{ et } c$$

Or, $\chi(g') = Opt(E') \leq \chi(o')$ donc

$$\chi(g) \leq \text{expr. de } \chi(o') \text{ et } c = \chi(o) = Opt(E)$$

Et par minimalité de $Opt(E)$, on a donc $\chi(g) = Opt(E)$.

Ce principe est décrit sur le schéma suivant :




# V Ordonnancement de tâches

## V.1 Description

On considère ici un problème voisin du problème précédent. On considère $n$ tâches $T = \{t_1, \ldots, t_n\}$ prenant une unité de temps pour être traitées sur une unité de calcul.

Chaque tâche $t$ dispose d'une date limite $f(t) \in [\![1, n]\!]$ **(deadline)** à laquelle elle doit être traitée sans quoi on écope d'une pénalité $p(t) \in \mathbb{N}$.

On appelle stratégie d'ordonnancement une application $d : T \to [\![0, n-1]\!]$ qui associe à chaque tache un unique temps de début $d(t)$. Selon cette stratégie, on déduit une séparation de $T$ en deux ensembles disjoints :

- $T^+(d)$ l'ensemble des tâches traitées dans les délais : $t \in T^+(d) \iff d(t) < f(t)$.
- $T^-(d)$ l'ensemble des tâches traitées en retard : $t \in T^-(d) \iff d(t) \geq f(t)$.

On note alors $P(d) = \sum_{t \in T^-(d)} p(t)$ la somme des pénalités des tâches en retard.

**Exemple** On considère l'ensemble de tâches :

| $t_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $f(t_i)$ | 1 | 2 | 3 | 4 | 4 | 4 | 6 |
| $p(t_i)$ | 3 | 6 | 4 | 2 | 5 | 7 | 1 |

Une stratégie d'ordonnancement (les tâches en retard sont en gras) est donnée dans le tableau suivant :

| $t_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $d(t_i)$ | **6** | 0 | 1 | **4** | 3 | 2 | 5 |

On a alors $P(d) = 5$.

On cherche à obtenir une stratégie d'ordonnancement de valeur $P(d)$ minimale.

On remarque que l'ordonnancement des tâches en retard n'a aucune importance, et on peut donc se contenter de déterminer une stratégie d'ordonnancement pour les tâches traitées dans les délais et la compléter par n'importe quel ordonnancement des autres tâches. On peut ainsi reformuler le problème : déterminer un sous-ensemble $T^+ \subset T$ de tâches **pouvant** être traitées dans les délais tel que $\sum_{t \in T^+} p(t)$ soit **maximale**.

## V.2 Algorithme glouton et implémentation

On résout maintenant ce problème de maximisation des pénalités $T^+$ par un algorithme glouton :

- On commence avec $T^+ = \emptyset$ et tous les temps de $[\![0, n-1]\!]$ sont marqués comme étant disponibles.
- On parcourt les tâches dans l'ordre décroissant des pénalités.
  - ⋆ Quand on considère la tâche $t$ s'il existe un temps $i$ disponible tel que $i < d(t)$ alors on marque comme indisponible le temps $i_0 = \max \{ i \in [\![0, n-1]\!] \mid i < d(t) \text{ et } i \text{ disponible} \}$ et on rajoute alors $t$ à $T^+$ en commençant $t$ au temps $i_0$.
- On place les tâches restantes aux temps disponibles.

Pour les structures de données, on utilise une représentation en tableaux de booléens (des `unsigned char` à `0` ou `1` en C) pour la disponibilités des temps. L'ensemble $T^+$ est alors implicite car il correspond aux tâches ordonnancé dans la première étape. Utiliser un tableau implique qu'une recherche linéaire soit faite pour chercher un plus grand temps disponible, et donc, la complexité temporelle globale sera en $O(n^2)$.

**Remarque** Il est possible d'améliorer cela pour passer en $O(n \log_2 n)$ (*exercice*).

Le programme C suivant implémente cet algorithme.

C
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    unsigned int id;
    unsigned int date_limite;
    unsigned int penalite;
    int debut; /* -1 tant que la tâche n'est pas ordonnancée */
} tache;

int compare_taches(const void *t1, const void *t2)
{
    return ((tache *)t2)->penalite - ((tache *)t1)->penalite;
```

```c
}


void *ordonnancement(tache *taches, size_t nb_taches)
{
    unsigned char *temps_occupe = malloc(sizeof(unsigned char) * nb_taches);
    memset(temps_occupe, 0, nb_taches);

    /* tri des activités par ordre décroissant des pénalités */
    qsort(taches, nb_taches, sizeof(tache), compare_taches);

    /* T+ par algorithme glouton */
    for (size_t k = 0; k < nb_taches; k++)
    {
        int i0 = -1;
        for (size_t i = 0; i < nb_taches; i++)
        {
            if (temps_occupe[i] == 0 && i < taches[k].date_limite)
                i0 = i;
        }
        if (i0 >= 0)
        {
            taches[k].debut = i0;
            temps_occupe[i0] = 1;
        }
    }

    /* Complétion par les tâches en retard */
    int i = 0; // indice du dernier temps disponible utilisé

    for (size_t k = 0; k < nb_taches; k++)
    {
        if (taches[k].debut == -1)
        {
            while(temps_occupe[i] == 1)
                i++;
            taches[k].debut = i;
            temps_occupe[i] = 1;
        }
    }

    free(temps_occupe);
}

int main()
{
    tache taches[] = {
        { 1, 1, 3, -1 }, { 2, 2, 6, -1 }, { 3, 3, 4, -1 },
        { 4, 4, 2, -1 }, { 5, 4, 5, -1 }, { 6, 4, 7, -1 },
        { 7, 6, 1, -1 }
    };

    size_t nb_taches = sizeof(taches) / sizeof(tache);

    ordonnancement(taches, nb_taches);

    for (size_t i = 0; i < nb_taches; i++)
    {
        printf("T%d (f:%d,p:%d) @ %d\n", taches[i].id, taches[i].date_limite,
                taches[i].penalite, taches[i].debut);
    }

    return 0;
}
```

Il produit la sortie :

```
T6 (f :4,p :7) @ 3
T2 (f :2,p :6) @ 1
T5 (f :4,p :5) @ 2
```

```
T3 (f :3,p :4) @ 0
T1 (f :1,p :3) @ 4
T4 (f :4,p :2) @ 6
T7 (f :6,p :1) @ 5
```

Ce qui correspond à l'ordonnancement $t_3, t_2, t_5, t_6, t_1, t_7, t_4$. Les tâches $t_1$ et $t_4$ sont en retard, donc la pénalité totale est de 5.

## V.3 Preuve d'optimalité

On va montrer que cet algorithme glouton renvoie un ensemble $T^+$ optimal. Pour cela, on procède comme précédemment. Tout d'abord, on montre qu'il existe une solution optimale qui effectue le premier choix de l'algorithme glouton.

**Théorème V.1** Soit $T$ un ensemble de tâches et $t \in T$ une tâche de pénalité maximale. Il existe un ensemble $T^+$ de tâches pouvant être traitées dans les délais, maximal pour les pénalités et tel que $t \in T^+$.

**Preuve**

Soit $T^+ \subset T$ un ensemble maximal. S'il contient $t$, il convient directement. Sinon, il existe une tâche $t'$ de $T^+$ qui est traitée à un moment où on pourrait traiter $t$ à temps (sinon $T^+ \cup \{t\}$ conviendrait et $T^+$ ne pourrait être maximal). On a $p(t') \leq p(t)$ par maximalité de $t$. L'ensemble $T'$ déduit de $T^+$ en remplaçant $t'$ par $t$ convient car on a forcément $P(T') = P(T^+)$ (en fait $\geq$ mais $=$ par optimalité de $T^+$) et par construction toutes ses tâches peuvent être traitées à temps.

■

On montre maintenant qu'en enlevant le choix glouton, on obtient une solution optimale du sous-problème.

**Théorème V.2** Soit $T^+ \subset T$ ensemble de tâches pouvant être traitées, maximal pour les pénalités et contenant une tâche $t$ de plus grande pénalité. Soit $i$ l'instant auquel la tâche $t$ commence dans un ordonnancement de $T^+$.

On pose $T' = T \backslash \{t\}$ avec des dates limites modifiées :

$$\forall t' \in T', d_{T'}(t') = \begin{cases} d_T(t') & \text{si } d_T(t') \leq i \\ d_T(t') - 1 & \text{sinon} \end{cases}$$

$T^+ \backslash \{t\}$ est alors maximal pour $T'$.

**Preuve**

Dans $T'$, on a à la fois enlevé $t$ et supprimé l'instant $i$. Tout ordonnancement de $T'$ peut alors être *relevé* en un ordonnancement de $T$ en décalant d'un instant les tâches commençant à partir de l'instant $i$ et en ordonnançant là la tâche $t$. Réciproquement d'un ordonnancement dans $T$, on déduit directement un ordonnancement de $T'$

Ainsi, s'il existait $T'^+$ maximal pour $T'$ tel que $P(T'^+) > P(T^+ \backslash \{t\}) = P(T^+) - p(t)$ alors $T'^+ \cup \{t\}$ serait de somme de pénalités strictement plus grande que celle de $T^+$ supposé maximal.

Donc, $T^+ \backslash \{t\}$ est maximal.

■

On conclut alors directement par récurrence sur le nombre de tâches comme on l'a fait précédemment pour la sélection d'activités :

**Théorème V.3** L'algorithme glouton renvoie un ordonnancement optimal.

# VI Exercices

Beaucoup de ces exercices sont des adaptations d'exercices du livre Algorithms de Jeff Erikson.

**Exercice 2** Un bibliothécaire souhaite ranger des collections de livres classées par auteur sur une longue étagère. On ainsi un suite $(a_1, \ldots, a_n)$ de collections données par la taille de la collection sur l'étagère, par exemple, en nombre de pages.

Un rangement des livres consiste à ordonner les collections de la première à la dernière sur l'étagère. Plus formellement, il s'agit d'une permutation $\sigma \in \mathfrak{S}_n$ où $\sigma(i)$ donne le numéro de la collection numéro $i$ dans

l'étagère.

Pour accéder à un auteur, il est nécessaire de parcourir linéairement l'étagère en partant de la première collection. Le coût d'accès à la k-ième collection est donc $cout(k) = \sum_{i=1}^{k} a_{\sigma(i)}$.

Le coût moyen d'accès aux collections est alors

$$\frac{1}{n}\sum_{k=1}^{n} cout(k)$$

Déterminer un algorithme permettant d'obtenir un rangement de coût minimal et l'analyser (correction, complexité).

**Exercice 3** Soit $X$ un ensemble de $n$ segments de $\mathbb{R}$. On dit que $Y \subset X$ couvre $X$ si $\bigcup_{y\in Y} y = \bigcup_{x\in X} X$. Le cardinal de $Y$ est appelé la taille de la couverture. On cherche ici une couverture de taille minimale.

a) Déterminer un algorithme glouton pour obtenir une couverture *a priori* de petite taille.

b) Renvoie-t-il une couverture de taille minimale ?

■ **Preuve**

a) On note $I_1 = [a_1; b_1], ..., I_n = [a_n; b_n]$ les segments. Pour réaliser un algorithme glouton, on va considérer le plus petit élément $x$ de $E = \bigcup I_i$. Comme ce sont des segments et qu'il y en a un nombre fini, ce plus petit élément existe toujours. Or, $x$ doit être couvert et pour cela, on va considérer le plus grand des segments le contenant. Comme c'est le minimum, il est forcément de la forme $I = [x; y]$. On pose alors $I'_i = \overline{I_i \backslash I}$ où

$$\overline{\emptyset} = \emptyset \quad \overline{]a;b]} = \overline{[a;b[} = \overline{[a;b]} = [a;b]$$

Comme $x$ ne peut être intérieur à un des $I_i$, la différence $I_i \backslash I$ est nécessairement un intervalle, donc d'une de ces quatre formes (le cas $]a; b[$ étant exclu car on retranche un segment à un segment). On a $E = I \cup \bigcup I'_i$ et on déduit une couverture de $\bigcup I'_i = I'_{i_1} \cup ... \cup I'_{i_p}$ par l'algorithme glouton (on omet alors les ensembles vides qui ne sont pas des segments mais ne contribuent pas à la couverture ou à $E$). Comme $I'_i \subset I_i$ on en déduit une couverture pour $E$.

b) Comme pour les autres preuves d'optimalité, on va la décomposer en deux temps.

- Tout d'abord, il existe une couverture optimale réalisant le choix glouton, c'est direct car une couverture optimale doit couvrir le minimum $x$ et elle contient ainsi un segment de la forme $I' = [x; y'] \subset [x; y] = I$ le plus grand segment. Donc on peut substituer $I$ à $I'$ tout en préservant la minimalité de la couverture.
- Ensuite, on raisonne par récurrence forte sur le nombre de segments pour montrer que le glouton est optimal. S'il n'y a qu'un segment, c'est direct. Si la propriété est vraie pour $1 \leq p < n$ segments et qu'on considère $n$ segments $I_1, ..., I_n$, alors on peut supposer que le choix glouton est $I_1$ quitte à réordonner les segments. Il existe donc un optimal $I_1 \cup I_{i_1} \cup ... \cup I_{i_r} = E$ et le glouton donne lui $I_1 \cup I_{j_1} \cup ... \cup I_{j_s} = E$ avec $1 + s \geq 1 + r$. On pose $I'_k = \overline{I_k \backslash I_1}$ et on a donc, par construction, $E \subset \bigcup I'_{j_k}$ et $E \subset I \cup \bigcup I'_{i_k}$. Or, la couverture $\bigcup I'_{j_k}$ est optimale par hypothèse de récurrence (on a au moins enlevé $I'_1 = \emptyset$) donc $s \leq r$. On en déduit directement que $s = r$.

■ **Note 1** La preuve peut être simplifiée tout en préservant son intérêt en ne considérant que des *segments* d'entiers $[\![a, b]\!]$. Ici, le problème vient du fait que si on a les segments $[1; 3]$ et $[2; 4]$, une fois $[1; 3]$ choisi, il faut couvrir $[2; 4]\backslash[1; 3] = ]3; 4]$ ce qui oblige à considérer $[3; 4]$.

■

■

**Exercice 4** On considère le problème du rendu de monnaie présenté dans le premier paragraphe.

a) Montrer que, pour le système $(b^0, b^1, \ldots, b^k)$ où $b \in \mathbb{N}, b \geq 2$ et $k \in \mathbb{N}$, l'algorithme glouton est optimal.

b) Déterminer un algorithme renvoyant le nombre de pièces optimal pour un système monétaire quelconque. *Indice : ce n'est certainement pas l'algorithme glouton.*

■ **Preuve**

a) Commençons par montrer que si $x \in \mathbb{N}^*$ tel que $b^i$ soit la plus grande pièce telle que $b^i \leq x$, alors il existe un rendu de monnaie optimal utilisant une pièce $b^i$. En effet, soit $(r_0, r_1, \ldots, r_k)$ un rendu optimal. S'il n'utilise pas la pièce $b^i$, alors il utilise uniquement les pièces $b^0$ à $b^{i-1}$. Comme $(b-1)\sum_{k=0}^{i-1} b^i = b^i - 1 < b^i \leq x$ il y a forcément une pièce $b^j$ avec $r_j \geq b$. Comme $r_j b^j = (r_j - b)b^j + b^{j+1}$, on peut réduire le rendu de

$b-1 \geq 1$ pièces ainsi. Comme il ne serait pas optimal, l'optimal utilise nécessairement la pièce $b^i$. On déduit alors par récurrence rapide que le glouton renvoie l'optimal. Ici, le fait qu'on soit forcé d'utiliser la pièce $b^i$ montre en fait qu'il n'y a qu'un unique rendu optimal. Celui-ci est, en vérité, l'écriture de $x$ dans la base $(b^0, b^1, \ldots, b^k)$ pour les nombres $< b^{k+1}$.

b) On remarque que $Opt(x)$, le nombre de pièces optimal pour $x$, vérifie, pour $x > 0$, $Opt(x) = 1 + \min_{p_i \leq x} Opt(x - p_i)$. On en déduit un algorithme *a priori* naïf consistant à résoudre cette récurrence en identifiant la pièce $p_i$ réalisant le minimum à chaque appel. On verra dans le chapitre sur la programmation dynamique qu'il est possible de rendre cet algorithme efficace en calculant ces minimums dans le bon ordre ou se souvenant des résultats des appels.

■

**Exercice 5** On considère un tableau $t = [a_0, \ldots, a_{n-1}]$ de $n$ entiers relatifs. Un couple $(i, j)$ est appelé un **segment positif pour** $t$ si $1 \leq i \leq j \leq n$ et $\sum_{k=i}^{j} a_k \geq 0$.

On dit qu'une suite de segments positifs couvre $t$ si chaque élément positif de $t$ a son indice dans un des segments.

Si on considère le tableau :

```
t = [| 3; -5; 7; -4; 1; -8; 3; -7; 5; -9; 5; -2; 4 |]
```

Alors $(0, 4)$ (de somme $3 - 5 + 7 - 4 + 1 = 2$), $(6, 8)$ (de somme $3 - 7 + 5 = 1$) et $(10, 12)$ (de somme $5 - 2 + 4 = 7$) est une suite de 3 segments couvrant $t$.

Si $t$ ne contient que des valeurs négatives, on considère qu'il est couvert par la suite vide 0 segment.

Déterminer un algorithme permettant d'obtenir une suite couvrant $t$ de plus petit cardinal et analyser celui-ci.

**Exercice 6** On considère le processus suivant : on part de l'entier 1 et à chaque étape on peut soit doubler l'entier courant, soit lui ajouter un. L'objectif est d'atteindre un entier cible $n$.

Par exemple, on peut atteindre 10 en quatre étapes ainsi :

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10$$

Tout entier peut être atteint par une succession d'incrément, mais ce n'est pas le plus court en terme de nombre d'étapes.

Déterminer un algorithme permettant d'obtenir le nombre d'étapes minimal pour atteindre un entier $n$ et l'analyser.

■ **Preuve**

On va considérer la stratégie suivante pour obtenir 1 en un minimum d'étapes en partant de $n$ :

- Si $n$ est pair, alors on le divise par 2
- Si $n$ est impair, on lui retranche 1.

Notons $G(n)$ le nombre d'étapes pour cette stratégie gloutonne. On a par construction $G(1) = 0$ et pour $p \geq 1$, $G(2p) = 1 + G(p)$ et $G(2p + 1) = 1 + G(2p) = 2 + G(p)$.

Le plus simple pour montrer que cette stratégie est optimale est de considérer le même type de formule pour l'optimal. En effet, $Opt(1) = 0$ et pour $p \geq 1$, $Opt(2p+1) = 1+Opt(2p)$ et $Opt(2p) = 1+\min(Opt(p), Opt(2p-1))$. On en déduit ainsi par une récurrence immédiate que $\forall n \geq 2, Opt(n) \leq 1 + Opt(n-1)$.

On va montrer par récurrence **forte** que $\forall n \geq 1, Opt(n) = G(n)$.

- Initialisation : on a vu que $Opt(1) = G(1) = 0$.
- Hérédité : Soit $n > 1$ tel que la propriété soit vérifiée pour tout entier $1 \leq p < n$. Par disjonction de cas selon que $n$ soit pair ou non :
  - ⋆ si $n = 2p + 1$, alors $G(2p + 1) = 1 + G(2p) = 1 + Opt(2p) = Opt(2p + 1)$
  - ⋆ si $n = 2p$, quitte à traiter à part le cas trivial $n = 2$, on peut supposer que $p > 1$. On a $G(2p) = 1+G(p)$ et $Opt(2p) = 1 + \min(Opt(p), Opt(2p - 1))$. Or

$$\begin{aligned} Opt(2p-1) &= G(2p-1) = 1 + G(2p-2) = 2 + G(p-1) \\ &= 2 + Opt(p-1) > 1 + Opt(p-1) \geq Opt(p) \end{aligned}$$

  Donc $Opt(2p) = 1 + Opt(p) = 1 + G(p) = G(2p)$.

Dans tous les cas, on a bien la propriété voulue.

■