Pour finir le chapitre sur les algorithmes gloutons, on s'intéresse en première partie à un dernier problème d'ordonnancement.

La deuxième partie contient deux exercices moins guidés, plus "style concours", que je vous incite à faire pour vous entraîner.

On travaillera cette fois en C, depuis la machine virtuelle ClefAgreg2019 et Emacs, ou depuis `https://www.onlinegdb.com/`.

On peut compiler (et exécuter) ce fichier avec `COMPILATEUR = gcc` ou `clang` comma ça :

```
$ COMPILATEUR -O3 -Wall -Werror -Wextra -Wvla -fsanitize=address \
  -fsanitize=undefined -pedantic -std=c11 -o TP24.exe TP24.c
$ ./TP24.exe
```

Problème 3 : *Scheduling with Deadlines*

Ce problème est assez similaire aux deux problèmes d'ordonnancement du TP23 précédent.

Description

On considère un entier $n \geqslant 2$ et un ensemble de n tâches $T = \{t_1, \dots, t_n\}$ (e.g. $T = \{1, \dots, 7\}$ comme dans l'exemple ci-dessous). Chaque tâche t_i prend une unité de temps (e.g., exactement une seconde) pour être traitées sur une unité de calcul.

Chaque tâche t dispose d'une *date limite* $f(t) \in [\![1, n]\!]$ (appelée *deadline* en anglais), à laquelle la tâche t doit avoir été traitée, sans quoi on doit payer une certaine *pénalité* $p(t) \in \mathbb{N}$.

On appelle *stratégie d'ordonnancement* une fonction $d : T \to \{0, \dots, n-1\}$ qui associe à chaque tâche $t \in T$ un unique temps de début $d(t)$. Évidemment, deux tâches différentes t_i et t_j doivent avoir un temps de début différent $d(t_i) \neq d(t_j)$.

Selon cette stratégie d, on en déduit une séparation de l'ensemble des tâches T en deux ensembles disjoints $T = T^+(d) \cup T^-(d)$:

— $T^+(d)$ est l'ensemble des tâches t traitées dans les délais : $t \in T^+(d) \Leftrightarrow d(t) < f(t)$ (t est commencée au temps $d(t)$ donc finie en $d(t)+1$ qui doit être $\leqslant f(t)$ sa *deadline*) ;

— $T^-(d)$ est l'ensemble complémentaire, autrement dit l'ensemble des tâches t traitées en retard : $t \in T^-(d) \Leftrightarrow d(t) \geqslant f(t)$ (finie en retard, après sa *deadline*).

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

Exemple

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

On donne cet ensemble de tâches t_i, avec leur *deadline* $f(t_i)$ et leur pénalité $p(t_i)$.
Une stratégie d'ordonnancement 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

— Q1. À quoi correspondent les tâches pour lesquelles $d(t_i)$ est noté en gras, dans ce tableau ?
— Q2. Calculer la pénalité totale $P(d)$ pour cette stratégie d'ordonnancement d.

Problème d'optimisation

Étant donné un ensemble de tâches T, et la donnée des *deadlines* $f(t_i)$ et des pénalités $p(t_i)$, on cherche à trouver une stratégie d'ordonnancement d, de valeur $P(d)$ **minimale**.

— Q3. Donner un exemple d'instance (simple et avec $n \geqslant 2$ petit) du problème telle que l'on puisse trouver une solution d qui ait une pénalité nulle.
— Q4. A contrario, donner un autre exemple d'instance telle que l'on ne puisse pas trouver de solution d avec une pénalité nulle. Quelle sera sa pénalité minimale ?

Ces deux autres exemples pourront être utiles pour tester l'implémentation, par la suite.

Et par "force brute" ?

A-t-on besoin d'un algorithme "intelligent" pour résoudre ce problème ? Autrement dit, peut-on s'en sortir avec une approche naïve d'exploration exhaustive, en temps raisonnable ?

— Q5. Calculer le nombre de solution $d : T \to \{0, \dots, n-1$ possible, sachant que cette application doit ordonnancer chaque tâche sur un temps de début unique (et que $|T| = n$).

  — En déduire une borne inférieure sur la complexité temporelle pire cas de l'algorithme "force brute" suivant :
  — On examine chaque ordonnancement possible d, pour lequel on calcule sa valeur $P(d)$, et on garde celui qui a la pénalité totale minimale.
  — On renvoie ce dernier à la fin.

Algorithme glouton

On peut commencer par remarquer un point intéressant : l'ordonnancement des tâches en retard (celles de $T^-(d)$) n'a aucune importance, et on peut donc se contenter de déterminer une stratégie d'ordonnancement d pour les tâches traitées dans les délais (celles de $T^+(d)$) et la compléter par n'importe quel ordonnancement des autres tâches.

On peut ainsi reformuler le problème : il faut déterminer un sous-ensemble de tâches $T^+ \subset T$ **pouvant être traitées dans les délais**, tel que soit maximale $\sum_{t \in T^+} p(t)$ soit **maximale**.

On va résoudre maintenant ce problème de maximisation des pénalités de T^+ par l'algorithme glouton que voici :

— On commence en posant $T^+ = \varnothing$ et tous les temps de l'ensemble $\{0, \dots, n-1\}$ sont marqués comme étant disponibles ;

— On parcourt ensuite les n tâches, dans un certain ordre (à préciser par la suite) :

  — Quand on considère la tâche t, s'il existe un temps i disponible, tel que $i < f(t)$, alors on marque comme indisponible le plus grand de ces temps possibles : $i_0 = \max\{i \in \{0, \dots, n-1\}, i < f(t) \text{ et } i \text{ disponible}\}$, et on rajoute alors la tâche t à l'ensemble T^+, en la commençant au temps i_0 (i.e., $d(t) := i_0$) ;

— À la fin, on place les tâches restantes aux temps disponibles (elles sont dans $T^- = T \backslash T^+$ et donc leur ordonnancement relatif n'a pas d'importance).

Pour l'ordre des tâches, on peut en envisager plusieurs. L'ordre que nous choisirons sera par ordre *décroissant* des pénalités $p(t)$. En effet, il semble logique d'essayer de placer en premier dans T^+ les tâches ayant les plus fortes pénalités, car le problème a été réécrit comme cherchant à maximiser la somme des pénalités $\sum_{t \in T^+} p(t)$.

— Q6. Au brouillon, exécuter l'algorithme glouton précédent, avec cet ordre initial des tâches, pour l'exemple de la figure de la page précédente.

Implémentation – en C

Pour l'implémentation en C, on commencera bien sûr par importer les bibliothèques usuelles dont vous aurez besoin.

Pour représenter une instance du problème, on va définir une structure, appelée `tache`, qui contient les champs suivants :

— `unsigned int id` : un identifiant de la tâche t, qui peut par exemple être son numéro $1, \dots, n$ comme dans l'exemple étudié plus haut ;

— `unsigned int date_limite` : la date limite (*deadline*) $f(t) \in \mathbb{N}$;

— `unsigned int penalite` : la pénalité $p(t) \in \mathbb{N}$;

— et enfin `int debut` qui sera son temps de début $d(t)$, initialement placé à `-1` tant que la tâche n'est pas ordonnancée, puis modifié (une seule fois) quand on a trouvé le temps i_0 à laquelle l'ordonnancer (ou un autre temps dans le deuxième cas de l'algorithme glouton, pour les tâches qui ne seront pas dans T^+).

— Q7. Écrire cette structure. On pensera bien à faire un `typedef struct tache tache`, pour écrire les types des tâches comme juste `tache` et pas `struct tache`.

— Q8. Écrire une fonction `tache creer_tache(unsigned int id, unsigned int date_limite, unsigned int penalite)` qui crée un objet local (pas sur le tas, pas besoin de `malloc`) avec ces champs, et le renvoie.

— Q9. Dans la fonction `int main(void)`, créer les tâches de l'exemple de la page 1, `tache1` à `tache7`.

— Q10. De même, créer un tableau `taches` contenant ces 7 tâches de l'exemple. On utilisera bien-sûr la notation `tache taches[] = { tache1, ..., tache7 }`.

Représentation des temps disponibles et indisponibles Pour la représentation en C du reste des données utilisées dans l'algorithme glouton, on sait qu'il n'est pas facile en C de représenter un ensemble (ici T^+), alors qu'en OCaml on peut se contenter d'une `'a list` à laquelle on ajoute (en tête pour le faire en temps constant) les éléments les uns après les autres.

Pour représenter la disponibilité des temps $\{0, \dots, n-1\}$, on va utiliser un tableaux de booléens `indisponible`. Les temps seront tous marqués à `false` (disponible, donc *pas indisponible*) dans `indisponible`, avec une simple boucle `for`, au début de l'algorithme glouton.

Tri des tâches Pour trier les tâches, on va procéder un peu comme en OCaml quand on utilise les fonctions `Array.sort` (qui trie en place) ou `List.sort` (qui renvoie une nouvelle liste triée). En C dans la librairie standard, on dispose de la fonction `qsort` (qui contrairement à ce que son nom laisse penser, n'est pas obligatoirement implémentée avec un tri rapide – *quick sort* en anglais). Elle s'utilise de la manière suivante :

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

Elle trie par ordre croissant le tableau `taches`, qui contient `nb_taches` ($= n$) objets, tous de taille `sizeof(tache)` en mémoire, et qui sont comparables grâce à une fonction de comparaison `compare_taches`, *que l'on doit implémenter avant.*

Cette fonction `compare_tache(t1, t2)` a le même rôle que la fonction de comparaison `compare : 'a -> 'a -> int` en OCaml : elle vaut `0` si `t1 == t2`, `+1` si `t1 > t2` (pour le critère choisi), et `-1` sinon (si `t1 < t2`).

— Q11. Écrire une fonction `int compare_taches(const void *t1, const void *t2)` qui compare les deux tâches `t1` et `t2`, pour les trier par ordre de pénalité *décroissante.*

  — Les deux pointeurs doivent être donnés sous forme de pointeurs de type `const void *`, et donc il faut les "recaster" en types `tache *` avant d'extraire leur champ `penalite`.

  — Cela aura cette forme là : `unsigned int penalite_t1 = ((tache *) t1) -> penalite;`.

Algorithme glouton

— Q12. Implémenter l'algorithme glouton, dans une fonction de signature `void *ordonnancement(tache *taches, size_t nb_taches)`.

  — Cette fonction n'aura rien à renvoyer : quand elle décide d'ordonnancer la tâche t au temps $d(t)$, elle le fait en modifiant son champ `debut`.

  — A la fin, toutes les tâches doivent avoir reçu une valeur unique dans $\{0, \dots, n-1\}$ dans leur champ `debut`.

  — Attention à bien libérer la mémoire allouée sur le tas pendant l'algorithme (qui nécessite un tableau de n booléens, `indisponible`).

  — Le type de donnée `size_t` correspond à un type entier, non signé, 32 ou 64 bits, utilisé par la machine pour les tailles et les indices de tableau.

  — Pour les boucles internes sur les tâches, on utilisera des variables (par exemple `k` et `i`) de type `size_t`, pas `int`, pour ne pas avoir d'erreur lors des comparaisons `k < nb_taches`.

— Q13. Dans le `main`, après un appel à `ordonnancement`, on peut afficher le résultat trouvé sous la forme suivante :

```
Après l'algorithme glouton :
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
```

La forme générique d'une de ces lignes d'affichage peut être :

```
Tid (f:deadline, p:penalite) @ debut
```

— Q14. Écrire une fonction `unsigned int penalite_totale(tache *taches, size_t nb_taches)` qui calcule la pénalité totale des tâches en retard, et utiliser la pour afficher à la fin `main` la "qualité" de la solution trouvée par l'algorithme glouton.

  — **Attention**, il faudra compter dans la somme les tâches qui sont ordonnancées après leur deadline (telles que $d(t) \geqslant f(t)$), *mais aussi* celles qui sont éventuellement pas encore ordonnancées (telles que $d(t) = -1$ d'après la représentation choisie).

— Q15. Tester cette fonction sur le tableau de tâches de l'exemple de la page 1, avant et après l'appel à `ordonnancement`. On devrait trouver cela :

```
Avant résolution, pénalité totale = 28.
Après résolution, pénalité totale = 5.
```

— Q16. Si vous voulez, ajouter un affichage de la pénalité totale, à chaque étape de la boucle `for` principale de l'algorithme glouton, pour la voir diminuer au fur et à mesure.
Par exemple :

```
La tâche T0 est placée en d = 3, et la pénalité totale = 21.
La tâche T1 est placée en d = 1, et la pénalité totale = 15.
La tâche T2 est placée en d = 2, et la pénalité totale = 10.
La tâche T3 est placée en d = 0, et la pénalité totale = 6.
La tâche T6 est placée en d = 5, et la pénalité totale = 5.
```

— Q17. Tester l'algorithme avec d'autres instances du problème.

Complexité temporelle de l'algorithme glouton

— Q18. Quelle est la complexité temporelle totale de l'algorithme glouton, sachant que l'appel à `qsort` sur un tableau avec `n = nb_taches` tâches se fait en temps $\Theta(n \log_2(n))$.

Preuve d'optimalité

Prouvons que cet algorithme glouton renvoie toujours un ensemble T^+ optimal, en trois points : 1, 2 et 3.

1. Tout d'abord, on montre qu'il existe une solution optimale qui effectue le premier choix de l'algorithme glouton.

Lemme : Soit T un ensemble de tâches, et $t \in T$ une des tâches de pénalité maximale. Il existe alors un ensemble T^+ de tâches pouvant être traitées dans les délais, maximal pour la somme de ses pénalités, et tel que $t \in T^+$.

> *Preuve :* Soit $T^+ \subset T$ un ensemble maximal. S'il contient la tâche 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 $T^+ \cup \{t\}$ conviendra, et donc T^+ ne pourrait pas être maximal). On a aussi $p(t') \leqslant p(t)$, par maximalité de t. L'ensemble T' déduit de T^+, en remplaçant t' par t, convient car on a forcément $P(T') = P(T^+)$ (même pénalités totales), puisque l'on a $P(T') \geqslant P(T^+)$ et donc égalité, par optimalité de T^+. Et par construction, toutes les tâches de T' peuvent être traitées à temps. □

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

Lemme : Soit $T^+ \subset T$ un ensemble de tâches pouvant être traitées, maximal pour les pénalités, et contenant une tâche $t \in T^+$ de pénalité maximale. Soit i l'instant auquel commence cette tâche t dans un ordonnancement de T^+.

— On pose $T' = T \backslash \{t\}$, avec des dates limites modifiées : $\forall t' \in T', d_{T'}(t') = d_T(t')$ si $d_T(t') \leqslant i$, ou $= d\_T(t') - 1$ sinon.

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

> *Preuve :* Dans T', on a à la fois enlevé la tâche 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 ici la tâche t. Réciproquement, depuis un ordonnancement de 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. Ceci est absurde.
>
> Donc on conclut que $T^+ \backslash \{t\}$ est maximal. □

3. On conclut alors directement par récurrence sur le nombre de tâches, comme on peut le faire pour les autres problèmes d'ordonnancement.

Théorème : L'algorithme glouton renvoie un ordonnancement optimal, pour le critère de tri qui prend en premier les tâches de pénalités les plus grandes (i.e. pour le tri par pénalités décroissantes).

Deux autres problèmes que l'on peut résoudre par des algorithmes gloutons – Exercices supplémentaires

Pour ces problèmes, n'hésitez pas à m'envoyer un mail et je peux donner un indice ou une piste.

Si vous en travailler un ou deux, et que vous rédigez une solution, n'hésitez pas non plus à me l'envoyer, je le corrigerai.

Le bibliothécaire optimisant

Un (ou une) bibliothécaire souhaite ranger des collections de livres classées par auteur sur une longue étagère. On considère ainsi une suite $(a_1, \dots, a_n)$ de collections, données par la taille a_i de la collection i sur l'étagère, par exemple, en nombre de pages ou en centimètres.

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-e auteur-e, comme on ne les trie pas par ordre alphabétique, 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 $\text{cout}(k) = \sum_{i=1}^{k} a_{\sigma(i)}$.

Le *coût moyen d'accès* aux n collections est alors donné par $\frac{1}{n} \sum_{k=1}^{n} \text{cout}(k)$.

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

Un genre de "le compte est bon"...

On considère le processus suivant : on part de l'entier 1, et à chaque étape on peut soit doubler la valeur de l'entier courant, soit lui ajouter 1. L'objectif est d'atteindre un entier cible donné $n \in \mathbb{N}^*$.

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 $n \geqslant 1$ peut trivialement être atteint, par une succession de $n-1$ incréments (de $+1$ en $+1$), mais ce n'est pas forcément le plus court en terme de nombre d'étapes.

Mettre au point un algorithme permettant d'obtenir le nombre minimales d'étapes nécessaires pour atteindre un entier n, et l'analyser (correction).

