DEVOIR SURVEILLÉ VI

Informatique

Durée : 4 heures

Consignes :

- La candidate veillera à présenter ses idées et réponses partielles même si elle ne trouve pas la solution complète à une question.
- **L'usage du cours, de notes, d'une calculatrice ou de tout autre appareil électronique est strictement interdit.**
- La candidate attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si une candidate est amenée à repérer ce qui peut lui sembler être une erreur d'énoncé, elle le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'elle a été amenée à prendre.
- **Indiquez si vous prenez la version normale ou étoilée du DS.**

Respectez les noms de fonctions indiqués, sinon le testeur vous donnera 0.
De même, un fichier qui ne compile pas vaut 0.

Les sections sont indépendantes.

Ne retournez pas la page avant d'y être invités.

I - Cours

Exercice 1

greeDY : Set Cover

On considère le problème suivant :

- Entrée : n un entier, et $\mathcal{S}$ un ensemble de parties[i] de $[\![1;n]\!]$ tel que $\bigcup_{s\in\mathcal{S}} s = [\![1;n]\!]$.

  En d'autres termes, on donne en entrée n et des ensembles $s_0, s_1, \ldots$ dont l'union vaut $[\![1;n]\!]$. Ces ensembles ne sont **pas** forcément deux à deux disjoints.
- Sortie : une liste d'ensembles de $\mathcal{S}$ dont l'union vaut $[\![1;n]\!]$. On cherche à **minimiser** le nobre d'ensembles de cette liste.

  En d'autres termes, on doit recouvrir tout $[\![1;n]\!]$ en utilisant le moins d'ensembles possibles (parmi ceux donnés en entrée).

NB : on dit qu'un élément x est *recouvert* s'il appartient à l'un des ensembles que l'on a choisi. Par exemple, choisir $\{1;3;5\}$ permet de recouvrir 1, 3 et 5.

Voici un exemple d'entrée/sortie pour Set Cover :

- On peut donner en entrée $n = 4$ et $\mathcal{S}$ qui est composé des ensembles suivants : $\{1;2\}, \{1;2;4\}, \{3;2\}, \{1;4\}, \{1;3;4\}$. L'union de ces cinq ensembles vaut bien $[\![1;n=4]\!]$.
- On peut recouvrir tout $[\![1;n]\!]$ en n'utilisant que deux des ensembles données : par exemple, $\{1;2;4\}$ et $\{3;2\}$.

  C'est bien une solution optimale car on ne peut pas tout recouvrir avec un seul ensemble avec cette entrée, donc on ne peut pas faire mieux que deux ensembles.

Voici des propositions d'algorithmes gloutons. Pour chacune d'entre elles, proposez soit un contre-exemple à l'optimalité, soit une preuve de correction. Il est garanti que s'il existe une preuve de correction elle est de la forme lemme d'échange puis récurrence.
Si l'ordre de parcours des éléments est ambigü, vous pouvez départager les ex-aequo comme vous le souhaitez.

1. Prendre le plus petit ensemble qui contient 1. Puis, si 2 n'est pas encore couvert, le plus petit ensemble qui contient 2. Etc etc jusqu'à avoir tout couvert.
2. Parcourir les ensembles de $\mathcal{S}$ par cardinal croissant. Pour chacun de ces ensembles, le prendre s'il contient un élément qui n'est pas déjà recouvert par les ensembles précédents.
3. Idem, mais parcourir les ensembles par cardinal décroissant.
4. Commencer par prendre le plus gros ensemble. Puis tant que l'on a pas recouvert tout $[\![1;n]\!]$, prendre l'ensemble ayant le plus d'éléments non-encore couverts.

Bonus (**très** difficile) : l'un de ces gloutons, même s'il n'est pas optimal, renvoie une solution qui contient au plus $\log_2(n)+1$ fois le nombre minimal d'ensembles. Trouver lequel et le prouver.

i. « partie » ou « sous-ensemble »

Consignes

Les élèves prenant le sujet Normal doivent faire les exercices « L-trominos » et « Grattes-ciels ». Les élèves prenant le sujet étoilé doivent faire « L-trominos » et « Justifier ».

Si jamais vous ne parvenez pas à écrire un code, un pseudo-code sur papier pourra être valorisé.

Indication de difficulté : dans l'exercice trominos, l'énoncé sous-entend très fortement un algorithme simple de résolution. Cependant, le mettre en oeuvre demande de manipuler finement des indices. Les deux autres exercices sont moins techniques à coder (moins d'erreurs d'indices) mais peut-être plus difficiles conceptuellement.

De cela, je pense que pour la plupart d'entre vous les second exercices seront plus simples.

Je rappelle l'indication centrale en informatique : **Faites. Des. Dessins.** Et testez-les sur des petits exemples pour vérifier si vos dessins sont corrects.

II - L-trominos (OCaml)

Carreler sa salle de bain

Cet exercice se base sur le sous-dossier `L-trominos` .

Dans cet exercice, par *paver* on entend "recouvrir une surface à l'aide de formes données sans qu'elles ne se superposent".

On considère un damier dont les côtés sont de longueur $n = 2^k$ (c'est à dire que le damier fait $2^k \times 2^k$ cases), avec $n \geqslant 2$. Par la suite, même si on ne le précisera plus, tous les damiers auront un côté de longueur une puissance de 2.

On indice les cases comme dans une matrice carrée mathématique mais en commençant les indices à 0 : les indices des lignes vont de 0 à $n - 1$ inclus et ceux des colonnes également.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Figure VI.1 – Un damier $2^2 \times 2^2$ avec les coordonnées indiquées.

Les formes que l'on considère pour paver sont les L-trominos, c'est à dire des formes composées de trois cases agencées en forme de L (ou de coin). Les voici :

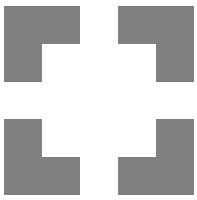

Figure VI.2 – Les 4 L-trominos.

1. Prouver qu'il est impossible de paver un damier avec des L-trominos.

On suppose dans la suite qu'il manque une case sur le damier :

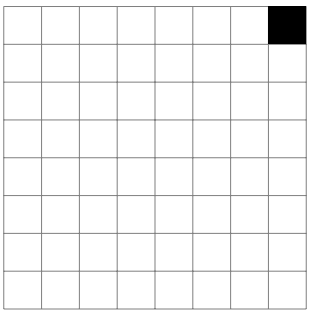

Figure VI.3 – Damier $2^3 \times 2^3$ auquel il manque l'angle $(0, 7)$.

L'objectif est de paver un tel damier, où une case est manquante. On travaillera en OCaml.

On représentera un L-tromino posé sur le plateau comme un 3-uplet qui contient les coordonnées des trois cases recouvertes par le tromino. Par exemple : `( (0,0), (1,0), (1,1) )` est un L-tromino posé sur l'angle en haut à gauche. Un *pavage* d'un damier (auquel il manque une case) est une liste de L-trominos qui le pavent.

On se donne donc les types suivants :

OCaml : solution.ml

```
(** Une case est la paire (i, j) de ses coordonnées *)
type case = int*int

(** Un tromino est un triplet de cases *)
type tromino = case * case * case

(** Un pavage est une liste de trominos *)
type pavage = tromino list
```

L'objectif est de construire un pavage d'un damier auquel il manque une case (peu importe où) ! L'idée est de découper le damier en 4 sous-damiers égaux. Il manque une case à l'un de ces sous-damiers : on peut donc le résoudre récursivement. Ensuite, on pose **un** L-tromino pour enlever une case aux 3 autres sous-damiers, et on les résout aussi récursivement.

Voici un schéma des étapes successives (les étapes 5 et 6 sont les mêmes, la seule différence et qu'en 6 on ne met plus en exergue une pièce particulière).

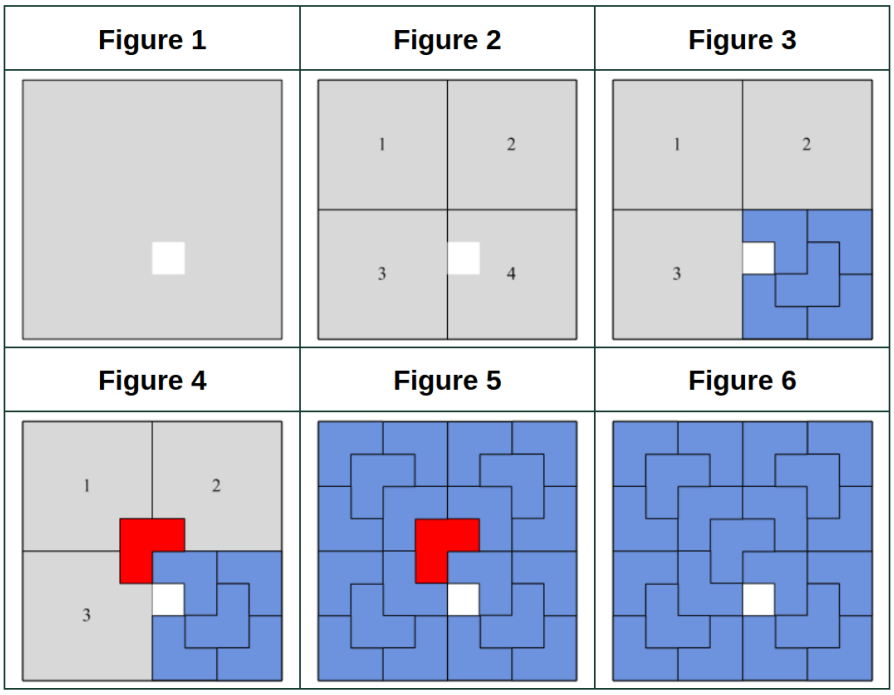

Figure VI.4 – Schéma de la solution

2. Appliquez la méthode ci-dessus pour paver le damier 8×8 auquel il manque la case $(0, 0)$.

J'attends un dessin du résultat final. Vous pouvez également dessiner des étapes intermédiaires si cela vous aide, en indiquant de quelles étapes intermédiaires il s'agit.
Au bout de 1h00 de DS (1h20 pour tiers-temps), vous pouvez lever la main et me demander de valider votre dessin, sans pénalité.

Afin de vous aider dans cet exercice, vous avez accès aux fonctions suivantes, que vous pouvez utiliser sans pénalité : :

— `est_piece : tromino -> bool` est une fonction qui prend en argument un triplet de paires d'entiers et s'évalue à `true` si elles forment un L-tromino et à `false` sinon.

— `check : case -> int -> case -> pavage -> bool` qui prend en enrée les coordonnées de l'angle en haut à gauche du damier, sa longueur, les coordonnées de la case manquante, une liste de tromino (aka un pavage) et vérifie si cette liste pave correctement ce damier.
Elle affiche en plus un petit message indiquant l'erreur s'il y en a une, ou vous félicitant s'il n'y en a pas.
Par exemple, `check (0,0) 2 (1,0) [(0,0), (1,0), (1,1)]` s'évalue à `false` et affiche une erreur car la case $(1, 0)$ est couverte deux fois : une fois car elle est manquante [1], et une fois car l'unique tromino de la liste la recouvre. (Dans cet exemple, on note en outre que $(0, 1)$ n'est pas couverte et que donc le pavage ne recouvre pas le damier.)

— `print_cases_couvertes : case -> int -> case -> pavage -> unit` qui prend les mêmes arguments que la fonction précédente. Elle renvoie une visualisation de la matrice qui compte combien de fois chaque case est couverte (la case manquante est considérée comme couverte de base). Elle peut vous aider à détecter quelles cases sont couvertes plusieurs fois.

— Un `Makefile`. Comme dans le DM, lancez `make exec` pour compiler et `make run` pour compiler et lancer le programme.

3. Donner les coordonnées des 4 cases centrales d'un damier $n \times n$ dont l'angle en haut à gauche est `(x0,y0)`.

4. Écrire une fonction `pavage : (int*int) -> int -> (int*int) -> pavage` qui prend en argument :
— En premier, une paire d'entier qui indique les coordonnées du coin en haut à gauche du damier/carré que l'on considère.
— En second, la longueur du côté de ce damier.
— En troisième, les coordonnées de la case manquante
Elle doit renvoyer une liste de L-trominos qui est un du damier.

Vous utiliserez la méthode récursive précédemment expliquée. Votre code risque d'avoir une disjonction à 4 cas qui se ressemblent beaucoup : c'est normal.
Vous pouvez utiliser la fonction `List.append` pour concaténer deux listes : `List.append l0 l1` renvoie la liste composée de tous les éléments de `l0` puis de tous ceux de `l1`.

5. (Vous pouvez faire cette fonction même sans avoir codée la précédente) Quelle est la complexité de la méthode récursive proposée ? On pourra pour simplifier faire comme si le fait de placer un tromino se fait en temps constant [2]

On suppose maintenant qu'il manque *deux* cases, en diagonale l'une de l'autre :

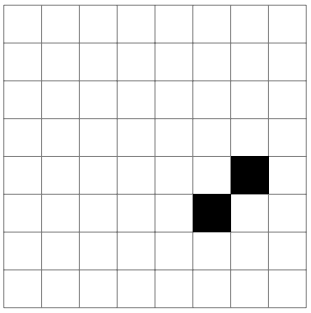

Figure VI.5 – Damier $2^3 \times 2^3$ auquel il manque les cases $(6, 6)$ et $(5, 7)$.

On n'utilise plus de L-trominos, mais des dominos (c'est à dire les formes composées deux cases adjacentes). Les voici :

1. Donc elle est « couverte par du vide ».
2. Selon comment vous avez programmé, c'est peut-être réellement le cas, ou peut-être pas.

Figure VI.6 – Les 2 dominos.

6. Est-il possible de paver un damier où il manque deux cases en diagonale à l'aide de dominos ? Si oui donner une façon de construire un pavage, si non le prouver.

III - Grattes-ciel (C, Normal)

Pour vous aider dans cet exercice, pour toutes les fonctions demandées sauf la dernière vous pouvez utiliser `nom_de_la_fonction_solution` qui est une version correcte de cette fonction que vous pouvez utiliser pour débuguer. Vous disposez également d'un `Makefile` : lancez `make run` pour compiler et lancer votre programme, `make exec` pour uniquement le compiler. Vous pouvez modifier la ligne de `CFLAGS` dans le `Makefile` pour changer les options de compilation ; la prochaine compilation devra alors être faite avec `make -B` .

On considère n immeubles en bord de l'océan. Le premier immeuble est au bord de l'océan. Le second est juste derrière le premier (il y a donc le premier entre l'océan et lui). Etc etc. On note $h_0, h_1, ..., h_{n-1}$ la hauteur de ces immeubles.

Voici un petit schéma de la situation :

```
OCEAN --- immeuble h0 --- immeuble h1 --- immeuble h2 --- ... --- immeuble hn-1
```

Voici un exemple avec $n = 5$, $h_0 = 1$, $h_1 = 3$, $h_2 = 2$, $h_3 = 3$, $h_4 = 5$:

```
                    |   |               |   |
                    |   |     |   |     |   |
  OCEAN --- | 1 | --- | 3 | --- | 2 | --- | 3 | --- | 1 |
```

On dit qu'on peut voir 'océan depuis un immeuble i (de hauteur h_i) si tous les immeubles entre lui et l'océan sont strictement plus petit, c'est à dire si pour tout $0 \leq j < i$, $h_j < h_i$.

Dans l'exemple précédent, peuvent voir l'océan : h_0 et h_1. Cependant, si on rasait h_1, alors h_0, h_2, h_3 pourraient tous trois voir l'océan. C'est le problème que l'on va résoudre : étant donnés des immeubles en bord d'océan, quels immeubles faut-il raser pour maximiser le nombre d'immeubles pouvant voir l'océan ?

On notera tout de suite qu'il est équivalent de trouver quels sont les immeubles à raser et quels sont les immeubles à garder. On va donc plutôt se concentrer sur ce second point. Les immeubles que l'on garde doivent former une suite strictement croissante (sinon ils ne peuvent pas tous voir l'océan). Et puisque l'on veut qu'il y ait le plus d'immeubles possibles qui peuvent voir l'océan, **on veut trouver une plus grande sous-suite strictement croissante de** $h_0, h_1, ..., h_{n-1}$.

En cas d'ex-aequo, on prendre n'importe quelle plus grande sous-suite strictement croissante.

En anglais, ce problème s'appelle *Longuest Increasing Subsequence* (LIS).

Par exemple, dans l'exemple précédent, h_0, h_2, h_3 est bien une suite strictement croissante et c'est même la seule plus grande sous-suite strictement croissante de h_0, h_1, h_2, h_3.

1. Proposer un exemple où il y a ex-aequo, c'est à dire qu'il y a au moins deux plus grandes sous-suites strictement croissantes différentes.

   Elles peuvent avoir des éléments en commun, mais elles doivent en avoir au moins 1 différent.

Pour simplifier, plutôt que de calculer la plus grande sous-suite, on commence par calculer sa longueur. On note $s(i)$ la longueur de la plus grande sous-suite strictement croissante se terminant à l'indice i.

Par exemple, toujours dans l'exemple précédent (on note entre parenthèses une sous-suite de longueur $s(i)$ et se terminant à l'indice i) :

- $s(0) = 1$ (h_0)
- $s(1) = 2$ (h_0, h_1)
- $s(2) = 2$ (h_0, h_2)
- $s(3) = 3$ (h_0, h_2, h_3)
- $s(4) = 1$ (h_4)

Grâce à tout cela, on conclut que la longueur d'une plus grande sous-suite est $s(3)$ c'est à dire 3.

2. Prouvez que $s(i)$ vérifie la formule de récurrence suivante :

$$s(i) = \begin{cases} 1 \text{ si } i = 0 \\ 1 + \max\{s(j) \mid 0 \leq j < i \text{ et } h_j < h_i\} \text{ sinon, si cet ensemble est non-vide} \\ 1 \text{ sinon} \end{cases}$$

On note que les deux derniers cas peuvent être algorithmiquement fusionnées en un seul :

Pseudo-code

```
s <- 1
Pour j allant de 0 à i exclu Faire :
  Si h_j < h_i Faire :.
    s <- max(s, 1 + s(i))
```

On considère que les h_i sont donnés par un pointeur `h` tel que `h[i]` $= h_i$.

3. Écrire une fonction `int s_naif(int const* h, int i)` qui calcule $s(i)$ à l'aide de cette formule de récurrence.

4. En déduire une fonction `int lis_naif(int const* h, int n)` qui calcule la longueur d'une plus longue sous-suite croissante de `h[0]`, `h[1]`, ..., `h[n-1]`.

5. Quelle est sa complexité ?

6. Montrer que des appels récursifs ont lieu plusieurs fois.

Pour optimiser, on stockera les $s(i)$ dans une zone mémoire assez grande pour contenir n entiers.

7. Écrire une fonction `int* cree_memoire(int n)` qui crée une zone mémoire assez grande pour contenir `n` entiers, initialise ces n entiers à -1, et renvoie l'adresse de cette zone.

8. Pourquoi initialiser à -1 ? Pourquoi cette valeur précise ?

9. Écrire une fonction `void libere_memoire(int* ptr)` qui libère une zone mémoire créée par la fonction précédente.

10. Écrire une fonction `int s_memo(int const* h, int i, int* mem)` qui prend en argument :

- `h` le pointeur vers les hauteurs des immeubles.
- `i` l'indice d'un immeuble
- `mem` une mémoire telle que pour tout indice $0 \leq j <$ `i`, si `mem[j] != -1` alors `mem[j]` $= s(j)$.

Votre fonction faire en sorte que $s(i)$ soit stocké dans `mem[i]` puis le renvoyer.

11. Déduire des fonctions précédentes une fonction `int lis_topdown(int const* h, int n)` qui calcule la longueur d'une plus longue sous-suite croissante de `h[0]`, `h[1]`, ..., `h[n-1]`.
Elle doit avoir une complexité temporelle quadratique en n. Vous ne devez pas faire de fuite de mémoire.

12. Prouver sa complexité.

13. Faire un schéma[3] indiquant les dépendances entre les $s(i)$: qui doit être calculé avant qui ? En déduire un ordre de calcul.

14. Pourriez-vous optimiser la mémoire utilisée en controlant l'ordre de calcul à la main ? Argumentez.

On veut maintenant afficher tous les immeubles à détruire. Pour chaque immeuble i à écrire, vous devez afficher la phrase `"Il faut raser l'immeuble i.\n"`. Vous pouvez les afficher dans l'ordre de votre choix.
Par exemple, sur notre bon vieil exemple, il faudrait afficher :

```
Il faut raser l'immeuble 2.
Il faut raser l'immeuble 4.
```

On aurait aussi pu afficher 4 puis 2 (ordre de notre choix).

15. Écrire une fonction `int plan_urbain(int const* h, int n)` qui effectue cet affichage, puis termine en renvoyant le nombre d'immeubles à détruire. Elle doit donc pour cela commencer par calculer une plus longue sous-suite strictement croissante.
On rappelle que l'on veut détruire tous les immeubles qui n'appartiennent pas à la plus longue sous-suite strictement croissante choisie.

3. oserais-je dire : un graphe !

IV - Justifier (C, Étoile)

Adapté de Mines-Ponts MP/PC/PSI, 2023

Pour vous aider dans cet exercice, pour toutes les fonctions demandées sauf la dernière vous pouvez utiliser `nom_de_la_fonction` qui est une version correcte de cette fonction que vous pouvez utiliser pour débuguer. Vous disposez également d'un `Makefile` : lancez `make run` pour compiler et lancer votre programme, `make exec` pour uniquement le compiler. Vous pouvez modifier la ligne de `CFLAGS` dans le `Makefile` pour changer les options de compilation ; la prochaine compilation devra alors être faite avec `make -B` .

On dit que l'on *justifie* un paragraphe lorsqu'on l'on aligne ses mots sur les bords gauche et droit de la zone de texte et que l'on place des espaces de sorte à ce que l'espacement entre les mots soient réguliers.

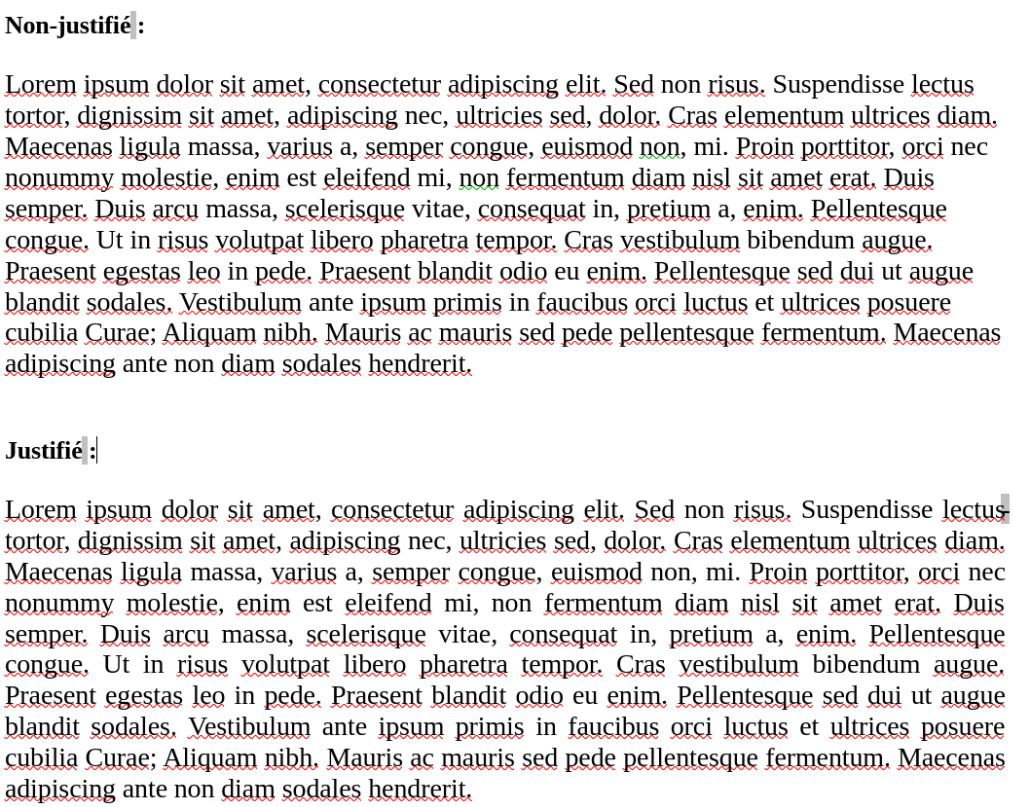

Figure VI.7 – Capture d'écran de LibreOffice montrant la différence entre texte non-justifié et justifié.

Le texte utilisé dans cette capture d'écran est le Lorem Ipsum, un texte pseudo-latin (qui ne veut rien dire) qui a la propriété d'avoir des enchainements de longueurs de mots qui ressemblent aux enchainements que l'on trouve dans les langues latines. Donc si une mise en page fait en sorte que le Lorem Ipsum « rende bien », alors elle « rendra bien » sur une langue latine quelconque.

Notre but est de justifier un texte. On note L le nombre de caractères (dont espaces, mais sans compter le `'\n'`) maximal que peut contenir une ligne. Notons qu'une fois que l'on connait les mots mis sur une lettre, il suffit de compléter les emplacements restants par des espaces (en en plaçant à peu près autant entre chaque mot). Autrement dit, le problème est de savoir quels mots mettre sur quelle ligne, donc où placer les retours à la ligne.

On ne s'autorise pas à réordonner les mots du texte.

On considère en entrée un ensemble de mots `char const** texte` ainsi que l'ensemble de leurs longueurs `int* len` (tel que `len[i]` est la longueur du mot `texte[i]`). On note `n` le nombre de mots.

On suppose que chaque mot peut tenir sur une ligne, i.e. que max `len` $\leq L$.

Pour mesurer si une ligne est harmonieuse, on utilisera la formule ci-dessous. i est l'indice du premier mot de la ligne, et j du dernier (inclus) :

$$cout(i,j) = \begin{cases} (L-(j-i)-\sum_{k=i}^{j} len[k])^2 \text{ si } \sum_{k=i}^{j} len[k] + (j-i) \leq L \\ +\infty \text{ sinon} \end{cases}$$

1. Expliquer à quoi correspond cette formule.

2. Écrire une fonction `int cout(int const* len, int L, int i, int j)` qui calcule $cout$.

   On utilisera `INT_MAX` pour coder $+\infty$

Dorénavant, on veut minimiser la somme des coûts des lignes.

1. On considère l'algorithme glouton suivant : « Ajouter autant de mots que possible à la première ligne. Puis autant que possible à la seconde. Etc ». Exhiber un exemple prouavnt qu'il n'est pas optimal.
2. Votre contre-exemple marche-t-il toujours si l'on enlève le carré du coût ? Que pensez-vous de l'effet de ce carré ?

On note $d(i)$ la meilleure somme des coûts des lignes pour agencer tous les mots `texte[i], texte[i+1, ..., texte[n-1]]` avec comme condition supplémentaire que `texte[i]` commence sa ligne. L'objectif est donc de réussir à calculer $d(0)$.

C'est à dire que l'on considère le texte `texte[i], texte[i+1, ..., texte[n-1]]`, que l'on place dedans des retours à la ligne de manière optimale, et que l'on se demande quel est le coût total associé.

3. Complétez et prouvez la formule suivante :

$$d(i) = \min_{i<j\leq n} (d(j) + ????)$$

   On pose comme cas de base $d(n) = 0$.
   Au bout de 1h00 de DS (1h20 pour tiers-temps), vous pouvez lever la main et me demander de valider votre formule, sans pénalité.
4. Écrire une fonction récursive `int d_naif(int const* len, int n, int L, int i)` qui calcule $d(i)$.
5. Montrez que certaines instances sont appelées plusieurs fois.
6. Écrivez une version mémoïsée de haut en bas `int d_topdown(int const* len, int n, int L)` qui calcule $d(0)$.
7. Comparez la complexité de vos deux fonctions. Commentez.
8. Si vous n'avez fait mémoïsé qu'une seule chose dans `d_topdown`, expliquez comment l'accélérer encore en mémoïsant autre chose. Puis modifiez votre fonction pour faire cette optimisation.
   Évaluez son gain de complexité.
9. Faites un schéma [4] qui indique l'ordre dans lequel les $d(i)$ sont calculés. En déduire un ordre de calcul.
   Dérécursifier votre fonction permettrait-il un gain d'ordre de grandeur de complexité spatial ? Pourquoi ?
10. Écrire une fonction `void justifie(char const** texte, int const* len, int L)` qui affiche le texte donné en argument, justifié.
    Vous utiliserez les méthodes précédentes pour trouver un découpage optimal sur plusieurs lignes. Au sein d'une ligne, vous répartirez les espaces de la manière la plus équilibrée possible. Vous penserez à aligner le premier mot d'une ligne à gauche, et le dernier mot d'une ligne à droite.

4. Oserais-je dire un graphe ?