TRAVAUX PRATIQUES IV

Des livres en pagaille

Ce TP contient deux parties : d'abord une étude du problème du remplissage, ensuite un morceau de cours sur la notion de portée (et un complément sur les boucles `for`).

Vous êtes encouragé·e·s à écrire le code le plus clair possible. En particulier, la création de fonctions auxiliaires pertinentes est encouragée.

Quand une preuve de terminaison est demandée, on attend un invariant. Quand une preuve de complexité est demandée, on attend que la conclusion soit un $\Theta(\dots)$ (et pas uniquement une majoration $O(\dots)$).

Rendu

Ce TP est également un DM. Il contient des questions papier, et des questions pratiques. Les premières sont à rendre sur feuille pour le premier cours d'informatique de la rentrée, les secondes par mail à `antoine.domenech@ac-poitiers.fr` **avant** ce premier cours.

Ma correction des codes utilise des tests automatiques. Aussi, je vous demande de vous assurer que votre code compile. Dans le cas contraire, la correction automatique ne vous donnera pas de points. Je vous encourage également à tester vos codes vous-mêmes.

J'évaluerai également la non-présence d'avertissements.

Mes mails sont ouverts pour toute question, y compris du type « je bloque, SOS ».

A Remplissage de bibliothèque

A.1 Présentation du problème

Considérez un rayon de bibliothèque : il dispose d'une longueur [1] d'étagère L sur laquelle on peut stocker des livres. Un livre est décrit par son épaisseur e. On dispose d'une famille de N livres $(e_i)_{0\leq i<N}$.

Informatiquement, les longueurs sont représentées par des `unsigned`. La famille de livres est représentés par un tableau contenant les e_i dans ses N premières cases [2].

Dans tout cet énoncé, sauf mention contraire, L est une longueur de rayon, N un nombre de livres et $(e_i)_{0\leq i<N}$ une famille d'épaisseurs de livres.

De plus, sauf mention contraire, J est un sous-ensemble de $[\![0;N[\![$ et $(f_j)_{j\in J}$ est une sous-famille de $(e_i)_{0\leq i<N}$ définie par pour tout $j\in J$, $f_j = e_j$.

On considère d'abord le problème Stockabilité qui consiste à savoir si les livres sont stockables sur le rayon :

- Entrée : L un entier strictement positif ; N une longueur de tableau et un tableau E (de longueur N) d'entiers positifs.
- Question : La somme des éléments du tableau est-elle inférieure ou égale à L ?

1. **a.** Écrire une fonction `bool est_stockable(unsigned L, unsigned N, unsigned E[10000])` qui résoud le problème Stockabilité. Elle devra renvoyer `true` si la somme des longueurs des livres de `E` est inférieure ou égale à `L`, et `false` sinon. Elle ne doit pas avoir d'effets secondaires.

b. Prouver sa terminaison. Quelle est sa complexité temporelle ?

2. On représente une sous-famille $(f_j)_{j\in J}$ de (e_j) en donnant J. Pour ce faire, on représente J comme un tableau de booléens J_tab tels que pour tout $i\in[\![0;N[\![$, $J\_tab[i]$ est `true` si et seulement si $i\in J$.

Autrement dit, J_tab est un tableau qui dit quels indices sont gardés pour construire la sous-famille, et lesquels sont rejetés.

Écrire une fonction :

```
bool sf_est_stockable(unsigned L, unsigned N, unsigned E[10000], bool J_tab[10000])
```

1. Avis aux physiciens homogénistes : disons que ce sont des `cm`. Ou des `mm`. Whatever, ce sont des `unsigned`.
2. On pourra dans le code faire comme si $N\leq 10000$ et donc utiliser des tableaux à 10000 cases.

qui teste si la sous-famille (f_j) est stockable, c'est à dire si $\sum_{j\in J} f_j \leq L$

Dans la suite, on s'intéresse au problème suivant : étant donnés L et $(e_i)_{0\leq i<N}$, on veut trouver le nombre maximal de livres que l'on peut stocker, c'est à dire la *taille maximale* d'une sous-famille $(f_j)_{j\in J}$ de (e_i) (où donc $J \subseteq [\![0; N[\![$) telle que $\sum_{j\in J} f_j \leq L$.

C'est un problème d'optimisation.

3. On suppose les e_i sont triés par ordre croissant. Soit $(f_j)_{j\in J}$ une sous-famille stockable qui est maximale pour son nombre de livres, c'est à dire pour $Card(J)$ (i.e. (f_j) est une façon de stocker le plus de livres possibles). Notons $\alpha = Card(J)$ son nombre de livres.

Prouver que la sous-famille $(e_k)_{k\in[\![0;\alpha[\![}$ est stockable. Que vient-on de prouver ?

4. **a.** En déduire une fonction `unsigned bibli_opti_triee(unsigned L, unsigned N, unsigned E[10000])` qui doit :

- Entrées : L un entier strictement positif ; N une longueur de tableau et E un tableau de N entiers positifs (triés par ordre croissant).
- Sortie : la taille maximale d'une sous-famille $J \subseteq [\![0; N[\![$ telle que $\sum_{j\in J} E[j] \leq L$.
- Effets secondaires : aucun.

b. Prouver la terminaison de votre fonction. Quelle est sa complexité temporelle ?

Et voilà un problème d'optimisation résolu, assez efficacement en plus ! Bémol toutefois : on a supposé E trié. Pour appliquer cette solution en toute généralité, il faudrait savoir trier un tableau…

B Tri d'un tableau

Cette sous-section contient deux versions : une version normale, et une version étoilée. Vous n'avez besoin d'en faire qu'une des deux.

B.1 Version normale : tri bulle

L'algorithme du tri bulle est un algorithme de tri plutôt simple. Il utilise la fonction suivante :

Pseudo-code

```
1  BULLAGE
2    Entrée : T un tableau, N sa longueur
3
4    Pour i allant de 0 inclus à N−1 exclu Faire :
5      Si T[i+1] < T[i] Faire
6        Échanger T[i] et T[i+1]
7
8    FIN
```

Et fonctionne ainsi :

Pseudo-code

```
1  TRI-BULLE
2    Entrée : T un tableau, N sa longueur
3    Sortie : aucune
4    Effets secondaires : trie T par ordre croissant
5
6    N fois, Faire :
7      BULLAGE(T, N)
8
9    FIN
```

Et voilà, ça trie. Ce n'est pas magique : en fait, on peut remarquer que chaque `BULLAGE` fait remonter au bout du tableau la plus grande valeur qui n'a pas encore été remontée. C'est à dire qu'après le premier bullage, $T[N-1]$ est la valeur maximale du tableau. Après le second bullage, $T[N-1]$ est le maximum et $T[N-2]$ est le second maximum. Etc etc etc, après N étapes on a bien [3] les éléments dans l'ordre.

5. **a.** Implémenter une fonction `void tri(unsigned T[10000], unsigned N)` qui implémente le tri bulle.

3. Je vous rassure, très bientôt nous prouverons au lieu d'expliquer. Ici, au début de chaque itération la propriété suivante est vraie : « les i dernières cases du tableau sont les i plus grandes et sont triées ».

b. Quelle est sa complexité ?

Point culture : le tri bulle est l'un des algorithmes de tris les plus simples à écrire. Il connait donc une certaine popularité dans les concours de programmation où il faut recoder un tri [4]. Par contre, c'est aussi un des algorithmes de tri les plus lents, et n'est donc guère utilisé dans les autres contextes.

B.2 Version étoile : tri insertion

L'algorithme de tri par insertion est un algorithme de tri très naturel. C'est celui que je fais en triant mes cartes, par exemple.

Il consiste à couper le paquet en deux : une première partie triée (la gauche de ma main), et une seconde pas encore triée (la droite de ma main). Tant que le paquet n'est pas trié, je prends la première carte de la seconde partie, et je l'insère au bon endroit dans la première partie.

(Un peu) plus formellement, on peut donner le pseudo-code suivant :

Pseudo-code

```
1  TRI-INSERTION
2    Entrée : T un tableau, N sa longueur
3    Sortie : aucune
4    Effets secondaires : trie T par ordre croissant
5
6    i ← 0
7    Tant Que i < N Faire :
8      // Au début de chaque itération, les i premières cases sont
9      // triées par ordre croissant
10     Insérer T[i] à la bonne place parmi les i premiers éléments
11       (quitte à en déplacer certains)
12     i ← i + 1
13
14   FIN
```

L'insertion consiste donc à :

- Trouver l'indice j (avec $0 \leq j \leq i$) auquel $T[i]$ doit être placé.
- Stocker $T[i]$ quelque part. On note tmp cette valeur stockée.
- On doit maintenant faire de la place pour libérer l'indice j. Pour cela, pour tout indice k tel que $j \leq k < i$, on décale d'une case vers la droite $T[k]$ (donc la case $i-1$ est envoyée sur la case i, la $i-2$ est envoyée sur la $i-1$, etc). Notez que la valeur d'origine de $T[i]$ est perdue dans le processus, c'est pour cela qu'on l'a mémorisée.
  Attention à faire ces déplacements dans un ordre intelligent qui ne cause pas de perte de valeur autre que $T[i]$.
- Écrire tmp dans $T[j]$.

6. **a.** Implémenter une fonction `void tri(unsigned T[10000], unsigned N)` qui implémente le tri par insertion.

b. Quelle est sa complexité ?

Point culture : le tri par insertion n'est pas le tri le plus rapide en toute généralité. Il est par contre remarquablement efficace sur les tableaux de petites tailles. De fait, beaucoup d'implémentations du tri dans les différents langages consistent à utiliser un algorithme de tri rapide (*quicksort*), qui passe la main au tri par insertion quand le tableau à trier devient assez petit.

B.3 Retour à la bibliothèque

7. **a.** À l'aide du tri, écrire une fonction `unsigned bibli_opti(unsigned L, unsigned N, unsigned E[10000])` qui doit :

- Entrées : L un entier strictement positif ; N une longueur de tableau et E un tableau de N entiers positifs (non-nécessairement trié).
- Sortie : la taille maximale d'une sous-famille $J \subseteq [\![0; N[\![$ telle que $\sum_{j \in J} E[j] \leq L$.
- Effets secondairs : aucun.

b. Quelle est sa complexité ?

4. Ainsi que dans les cours d'informatique.

Interlude : l'algorithme que l'on utilise appartient à la grande famille des *algorithmes gloutons*. Ceux-ci consistent à construire une solution optimale d'un problème à l'aide d'une succession de choix. Une fois un choix effectué, on ne revient jamais dessus. Chaque choix doit être simple. Ici, une solution optimale est une famille (f_j) stockable la plus grande possible, et les choix successifs permettant de la construire consistent à prendre à chaque fois le livre disponible le moins épais.

Ces algorthimes ont l'avantage d'être simples à concevoir, efficaces (pour peu que leur choix soit efficace), et plutôt [5] simples à prouver corrects. Ils ont le désavantage d'être souvent incorrects.

On s'intéresse maintenant à la question de maximiser non plus le nombre de livres stockés, mais le taux de remplissage du rayon.

Étant donné L une longueur de rayon, $(e_i)_{0\leq i<N}$ une famille d'épaisseurs de livres et $(f_j)j \in J$ une sous-famille de livres stockable (que l'on va ranger sur le rayon), on définit le taux d'occupation η du rayon comme :

$$\eta = \frac{1}{L}\sum_{j\in J} f_j$$

Comme (f_j) est stockable, on a toujours $0 \leq \eta \leq 1$.
Autrement dit, le rayon est-il occupé à 100% ? 99% ? 1% ? Etc etc.

8. Si aucun livre n'est stockable, i.e. si pour tout i on a $e_i > L$, quel est le taux d'occupation maximal que l'on peut obtenir ?

Dans la suite, on suppose qu'il existe au moins un livre stockable.

9. Exhiber pour L et N quelconques [6] un exemple atteignant $\eta = 1$.

10. Exhiber pour L et N quelconques un exemple avec $\eta = \frac{1}{L}$. Peut-on atteindre un ratio non-nul inférieur ? Justifier.

11. Pour essayer de maximiser le ratio, on essaye de modifier notre algorithme glouton de la façon suivante : au lieu de toujours sélectionner le livre le moins épais, on sélectionne le livre le plus épais (pouvant encore être stocké).

On suppose qu'il existe au moins un livre stockable. Quel est le ratio minimal que cet algorithme peut atteindre ?

12. (Difficile) Trouver un algorithme polynomial qui résout le problème de maximisation du taux d'occupation, ou à défaut prouver qu'il n'en existe pas.

C Retour au tri

Un théorème célèbre sur les algorithmes de tri est le suivant :

Théorème 1

On considère une structure (par exemple un tableau) à n éléments. Ces éléments sont comparables, c'est à dire qu'on peut leur appliquer `<` .
Tout algorithme de tri qui fonctionne uniquement en comparant entre eux (et en les déplaçant) des éléments de la structure à trier fonctionne en temps $\Omega(n\log_2(n))$.

L'idée de la preuve est la suivante :

Trier un tableau consiste à réordonner ses élements. Il y a $n!$ façons possibles de réordonner les n éléments d'un tableau, et une seule qui est triée par ordre croissant. L'algorithme ne peut procéder qu'en faisant des comparaisons d'éléments. Lorsqu'il compare $T[i]$ et $T[j]$, il en déduit que certaines façons d'ordonner le tableau sont peut-être les bonnes (celles où la plus petite valeur parmi $T[i]$ et $T[j]$ est avant la plus grande), et que d'autres sont mauvaises.

Or, on peut montrer que le meilleur des cas consiste à éliminer la moitié des façons à chaque comparaison [7]. Il faut donc au moins $\Omega(\log_2(n!))$ comparaisons pour identifier la seule bonne façon de réordonner, c'est à dire $\Omega(n\log_2(n))$ comparaisons [8].

Le cas particulier des entiers est intéressant. Sur les entiers, on peut faire d'autres choses que comparer des éléments du tableau entre eux. Par exemple, si l'on sait que [9] tous les éléments du tableau sont compris dans un intervalle $[\![m; M]\!]$, on peut :

— Compter pour chaque valeur comprise entre m et M combien de fois elle apparait dans le tableau.

— En déduire le tri du tableau.

Cela donne le pseudo-code suivant :

5. Offre soumise à conditions.
6. i.e. votre exemple doit marcher pour n'importe lesquels, vous ne devez pas faire un exemple pour un L et/ou un N bien choisi
7. Pourquoi ? Parce qu'il faut que *dans le pire des cas*, on termine le plus rapidement possible. Or, ne pas éliminer la moitié des façons fait que dans le pire des cas il reste plus de la moitié des façons à tester. On ne peut "donc" pas faire mieux qu'éliminer la moitié à chaque fois.
8. Plus précisément, on a $\ln(n!) \sim n\ln n$. Le prouver est un exercice de maths. Personnellement je le fais avec la formule de Stirling ; mais je me souviens l'avoir fait sans en colle dans une vie antérieure.
9. Notez l'hypothèse qui nous permet d'échapper au théorème.

Pseudo-code

```
1  TRI-DÉNOMBREMENT
2    Entrée : T un tableau de N entiers
3    Sortie : aucune
4    Effets secondaires : trie T
5
6    Compte ← tableau indicés de m à M inclus
7    Initialiser les cases de Compte à 0
8
9    Pour i allant de 0 inclus à N exclu Faire :
10     val ← T[i]
11     Compte[val] ← Compte[val] + 1
12
13   i ← 0
14   Pour val allant de m inclus à M inclus Faire :
15     Répéter Compte[val] fois :
16       T[i] ← val
17       i ← i + 1
18
19   FIN
```

Soit T un tableau d'entiers non-signés, tous strictement inférieurs à 20000. On utilisera $m = 0$ et $M = 20000$.

13. Écrire une fonction `void tri_denomb(unsigned T[10000], unsigned N)` qui à l'aide d'un tri par dénombrement tri le tableau `T` (de longueur `N`).

14. Quelle est la complexité temporelle de votre fonction ? Commenter.

15. On ôte l'hypothèse "tous strictement inférieurs à 20000". On autorise également les entiers à être négatifs. On suppose que l'on peut créer un tableau dont la taille est une variable [10]. Comment appliquer le tri par dénombrement sur le tableau d'entiers ? Avec quelle complexité ?

On ne demande pas un code, mais une explication. Il s'agit surtout d'indiquer quels choix de m et M on fait, et les conséquences éventuelles sur la complexité.

16. Donner un défaut du tri par dénombrement, à même d'expliquer pourquoi il n'est pas si souvent [11] utilisé.

10. Nous verrons cela bien assez tôt. Teaser : `malloc` .
11. Je donnerais bien plus de contexte sur « pas si souvent », mais je ne veux pas divulgacher la réponse.

D Portée

D.1 Blocs et portées

Définition 1 : Bloc

En C, un bloc de portée est un morceau du code délimité par des accolades `{` `}` . Il contient une suite d'instructions, terminées par des point-virgule `;` . Un bloc de portée peut-être inclus dans un autre.
Le corps d'une fonction ou d'un `if` sont des exemples les plus courants de blocs de portée. Des `if` imbriqués fournissent un bon exemple de blocs de portés imbriqués.
Le corps d'un `while` ou d'un `for` sont aussi des blocs de portée, particuliers : un bloc est limité à une itération de la boucle. En effet, atteindre la fin d'une itération revient à atteindre une accolade fermante donc à fermer un bloc ; et en commencer une nouvelle revient à atteindre une accolade ouvrante donc à en commencer un nouveau.

Si ce dernier point vous perturbe, retenez-en uniquement de suite la conséquence : on ne doit pas *déclarer* une variable *dans* une boucle avec pour but de l'utiliser ensuite dans les itérations *suivantes* de cette *même* boucle.

Définition 2 : Portée

La portée d'un identifiant est l'ensemble des endroits du code où l'identifiant peut-être utilisé.
En C, la portée d'un identifiant commence à sa déclaration et se finit la fin du bloc de portée où elle a été déclarée.
La portée des arguments d'une fonction est le corps de la fonction. La portée d'une déclaration de compteur de boucle d'un `for` est cette même boucle `for`.

Ainsi, si une variable est déclarée dans un bloc, il ne sert à rien de l'appeler après le bloc : l'identifiant n'existe plus [12] !
Par exemple, le code suivant pour tester si un entier ets premier ne fonctionne pas, parce que l'identifiant `d` n'est plus accessible après la boucle (il fonctionnerait sinon) :

C

```c
/* Teste si n est premier */
bool est_premier(int n) {
  bool sortie = true;
  for (int d = 2; d < n; d = d+1) {
    if (n%d == 0) { // si d divise n
      sortie = false;
      break;
    }
  }
  if (d == n) { // donc si aucun d ne divisait n
    sortie = false;
  }
  return sortie;
}
```

Définition 3 : Masquage

Considérons un bloc d'un code, contenant un sous-bloc. Supposons qu'ils définissent tous deux un même identifiant P. Dans ce cas, il y a *masquage* : lorsque l'on est dans le sous-bloc, appeler P revient à appeler la variable du sous-bloc et non celle du bloc.
On dit aussi que l'identifiant du sous-bloc *masque* celui du bloc.

En cas de masquage, il est impossible d'appeler l'identifiant masqué. Ce n'est pas pratique. En outre, cela peut faire du code difficile à relire : on évitera donc les masquages.
Voici une exemple de masquage :

C

```c
int n = 3;
for(int i = 0; i < n; i = i+1) {
  for (int i = 0; i < n; i = i+1) {
    printf("%d␣", i);
  }
}
```

12. Notez que savoir si le *nom* est utilisable est différent de savoir si l'objet mémoire existe encore, i.e. si la donnée est encore écrite en mémoire. On y viendra plus tard en cours.

Ce code va afficher trois fois le texte `0 1 2` . Si vous ne comprenez pas pourquoi, faites le tourner à la main en distinguant bien les deux `i`.

D.2 Et les boucles `for` ?

Notez que l'on peut mettre un bloc dans un code où l'on veut. Par exemple :

C
```c
int demo(int x) {
  int s = 23;
  {
    int a = 42;
    s = 42+a;
  }
  return s;
}
```

Ce code implémente (sur les entiers) la fonction constante égale à 65.

On évite généralement de créer ainsi des blocs de portée sans raison. Toutefois, c'est important à garder en tête, car c'est ainsi que fonctionne un `for`. Le compilateur transforme le code suivant :

C
```c
avant;
for(init; condition; maj) {
  todo;
}
apres;
```

En :

C
```c
avant;
{ // notez les accolades !
  init;
  while (condition) {
    todo;
    maj;
  }
}
apres;
```

On retrouve bien ce que vous savez déjà [13] :

- Les variables déclarées dans l'initialisation d'un `for` sont locales au `for`. Cela est dû au bloc de portée défini par les accolades de la première et dernière ligne.
- Un `for` est juste une façon plus lisible d'écrire certaines boucles `while`.
- L'initialisation d'un `for` a lieu *avant* la boucle.
- Une itération de la boucle correspond à effectue le corps du `for` puis à mettre à jour les compteurs du boucle. Autrement dit, *la mise à jour du compteur d'un Pour a lieu à la fin d'une itération*, et non au début.

13. Et que je vous bassine car ce sera important en preuves de correction.