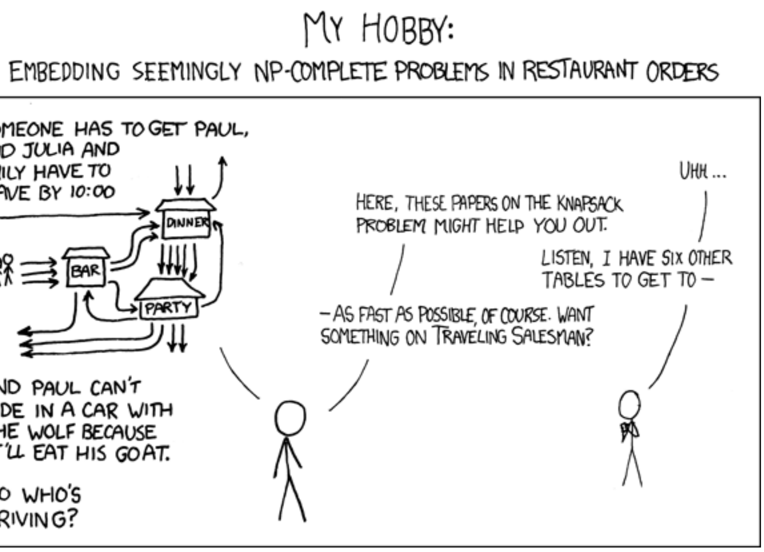
MPI* Info - TP8

Algorithmes probabilistes et d'approximation



O. Caffier

1 Algorithmes probabilistes

Question 1.1 Écrire une fonction `partition : int array -> int -> int -> int -> int`
Corrigé :

```
let partition(tab : 'a array) (debut : int) (fin : int) (pivot_ind : int) : int =
  let pivot = tab.(pivot_ind) in
  (* Echange le pivot avec le dernier elt pour simplifier le traitement *)
  let tmp = tab.(fin) in
  tab.(fin) <- tab.(pivot_ind);
  tab.(pivot_ind) <- tmp;

  let k = ref debut in
  for i = debut to fin - 1 do
    if tab.(i) <= pivot then
      begin
        let tmp = tab.(i) in
        tab.(i) <- tab.(!k);
        tab.(!k) <- tmp;
        incr k
      end
  done;

  (* Retablit le pivot a sa position correcte *)
  let tmp = tab.(fin) in
  tab.(fin) <- tab.(!k);
  tab.(!k) <- tmp;
  !k
```

Question 1.2 En déduire une fonction `random_quicksort : int array -> unit`
Corrigé :

```
let random_quicksort (tab : int array) : unit =
  let rec aux (debut : int) (fin : int) : unit =
    if debut < fin then begin
      let pivot_ind= debut + Random.int (fin-debut) in
      let pivot_new_pos = partition tab debut fin pivot_ind in
      aux debut (pivot_new_pos -1);
      aux (pivot_new_pos +1) fin
    end
  in aux 0 (Array.length tab -1)
```

Question 1.3 Écrire une fonction `test_product: int array array -> int array array -> int array array -> bool` **Corrigé :**

```
exception Diff_coord

let check_coordinates (tab1: int array array) (tab2 : int array array) : bool =
  (* On suppose les tab de mm dimension *)
  try
    for ligne = 0 to Array.length tab1-1 do
      for colonne =0 to Array.length tab1.(0) -1 do
        if tab1.(ligne).(colonne) <> tab2.(ligne).(colonne) then raise Diff_coord
      done;
    done;
    true
  with Diff_coord -> false


let produit_matrice (a : int array array) (b : int array array) : int array array =
  (* a est de taille m*n et b de taille n*p, le res de taille m*p *)

  let m = Array.length a in
  let p = Array.length b.(0) in
  let n = Array.length a.(0) in
  let res = Array.make_matrix m p 0 in
  for i = 0 to m -1 do
    for j = 0 to p -1 do
      (* Calcul du coeff *)
      for k = 0 to n-1 do
        res.(i).(j) <- res.(i).(j) + a.(i).(k)* b.(k).(j)
      done
    done done;
  res

let test_product (a : int array array) (b : int array array) (c : int array array) : bool =
  let p = Array.length b.(0) in
  let x_random = Array.init p (fun i -> let rand = Random.int 2 in [|rand|]) in
  let bx = produit_matrice b x_random in
  let cx = produit_matrice c x_random in
  let abx = produit_matrice a bx in
  check_coordinates abx cx
```

2 BinPacking

$\boxed{1}$ Donner une solution optimale pour BinPacking sur l'instance $C = 10$ et $X = 2, 5, 4, 7, 1, 3, 8$
Corrigé : Il vient $k = 3$ avec

$$\begin{aligned} B_0 &= \{2, 8\} \\ B_1 &= \{5, 4, 1\} \\ B_2 &= \{7, 3\} \end{aligned}$$

2.1 Caractère NP-complet

$\boxed{2}$ Définir le problème BPD associé au problème d'optimisation BinPacking
Corrigé :

BPD

Entrée : $C \in \mathbb{N}, X = x_0, \ldots, x_{n-1}, k \in \mathbb{N}$
Sortie : true ssi il existe une partition de X en $B_0 \sqcup \ldots \sqcup B_{p-1}$ tq $\forall i, \sum_{x \in B_i} x \leq C$ et $p \leq k$

$\boxed{3}$ Montrer que PARTITION est NP-complet.
Corrigé :
D'une part,

— On définit $v : X = \{x_0; x_{n-1}\}, < I > \longmapsto$ **true** $\Leftrightarrow \sum_{i \in I} x_i = \sum_{i \notin I} x_i$
— Le problème de décision associé à v est dans P
— $| < I > |$ est polynomiale en $\# X$
— Il vient

$$\begin{aligned} X \in \text{PARTITION} &\Leftrightarrow \text{il existe } I \subset [|0; n-1|] \text{ tq. } \sum_{i \in I} x_i = \sum_{i \notin I} x_i \\ &\Leftrightarrow \exists < I > \in \Sigma^* \text{ tq. } v(X, < I >) = \textbf{true} \end{aligned}$$

$\boxed{\text{d'où PARTITION } \in NP}$

D'autre part, considérons la transformation suivante :

$$\varphi : x_0; \ldots; x_{n-1}, s \longrightarrow x_0; \ldots; x_{n-1}, 2s, \sum_{i=1}^{n-1} x_i$$

— $\underline{\Rightarrow}$: Supposons que il existe $B \subset \{x_0; \ldots; x_{n-1}\}$ tq. $\sum_{x \in B} x = s$.

Notons $S = \sum_{i=0}^{n-1} x_i$. Ainsi,

$$\boxed{S + \sum_{x \in B} x = s + S}$$

et

$$\begin{aligned} \sum_{x \notin B} x &= \sum_{i=0}^{n-1} x_i - \sum_{x \in B} x \\ &= S - s \end{aligned}$$

En ajoutant $2s$ des deux côtés, il vient

$$\boxed{2s + \sum_{x \notin B} x = S + s}$$

D'où

$$\boxed{\boxed{S + \sum_{x \in B} x = 2s + \sum_{x \notin B} x}}$$

Ainsi, on prend

$$I = \{i \in [|0; n-1|] \mid x_i \in B\} \cup \{n+1\}$$
$$I^c = \{i \in [|0; n-1|] \mid x_i \notin B\} \cup \{n\}$$

Remarque : Ici, n et $n+1$ sont les indices de $2s$ et S lorsqu'ils sont considérés par le problème PARTITION.

— $\underline{\Leftarrow}$: Supposons qu'il existe $I \subset [|0; n+1|]$ tq. $\sum\limits_{i \in I} x_i = \sum\limits_{i \notin I} x_i$ (avec $x_n = 2s, x_{n+1} = S$). Notons $I' = I \cap [|0; n-1|]$.

  — 1er cas : $n, n+1 \in I$. Alors

$$S + 2s + \sum_{i \in I'} x_i = \sum_{i \notin I'} x_i$$

  or $2s + \sum\limits_{i \in I'} x_i > 0$ et $\sum\limits_{i \notin I'} x_i \leq S$.

ABS

  — 2ème cas : $n, n+1 \notin I$

$\rightarrow$ idem

  — 3ème cas : $n \in I, n+1 \notin I$ (l'autre cas est analogue).
  Ainsi,

$$S + \sum_{i \in I'} x_i = 2s + \sum_{i \notin I'} x_i$$
$$\text{d'où } S + \sum_{i \in I'} x_i = 2s + \sum_{i=0}^{n-1} x_i - \sum_{i \in I'} x_i$$
$$\text{d'où } S + 2\sum_{i \in I'} x_i = 2s + S$$

  Finalement,

$$\boxed{\boxed{\sum_{i \in I'} x_i = s}}$$

  On prend donc $B = I' \subset [|0; n+1|]$.

D'où le résultat voulu.

D'où SubsetSum $\leq_p$ PARTITION

Ainsi, PARTITION $\in NP$ et SubsetSum $\leq_p$ PARTITION (avec SubsetSum NP-complet) ce qui permet de conclure.

$\boxed{\mathbf{4}}$ En déduire que BPD est NP-complet.

Corrigé :

D'une part,

— On définit $v : C, X, < B_0, \ldots, B_{p-1} >, k \longmapsto \textbf{true} \Leftrightarrow \forall\, i, \sum_{x \in B_i}$ et $p \leq k$

— $| < B_0, \ldots, B_{p-1} > |$ est polynomiale en $\# X$

— Il vient

$$\begin{aligned} X, k \in BPD &\Leftrightarrow \ldots \\ &\Leftrightarrow \exists P = < B_0, \ldots, B_{p-1} > \in \Sigma^* \text{ tq. } v(C, X, P, k) = \textbf{true} \end{aligned}$$

Donc BPD $\in NP$

D'autre part, considérons la construction suivante

$$\psi : x_0, \ldots, x_{n-1} \longrightarrow X = x_0, \ldots, x_{n-1}, C = \lfloor \frac{\sum_{i=0}^{n-1} x_i}{2} \rfloor, k = 2$$

— $\underline{\Rightarrow}$: supposons qu'il existe $I \subset [|0; n-1|]$ tq $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$.

Alors, il est immédiat que

$$\begin{aligned} \sum_{i \in I} x_i &= \lfloor \frac{\sum_{i=0}^{n-1} x_i}{2} \rfloor \\ \text{d'où } \sum_{i \in I} x_i &\leq \lfloor \frac{\sum_{i=0}^{n-1} x_i}{2} \rfloor \end{aligned}$$

De même pour I^c. On a bien notre partition :

$$\begin{aligned} B_0 &= \{x_i \mid i \in I\} \\ B_1 &= \{x_i \mid i \not\in I\} \end{aligned}$$

$\rightarrow$ OK

— $\underline{\Leftarrow}$: Supposons qu'il existe une partition de X en $B_0 \sqcup B_{p-1}$ tq. $\forall\, i, \sum_{x \in B_i} x \leq C$ et $p \leq 2$.

  — $\underline{\text{1er cas :}}$ $p = 1$, impossible (car on ne respecterait pas notre inégalité avec la partie entière).

  — $\underline{\text{2ème cas :}}$ $p = 2$. On a

$$\begin{cases} \sum_{x \in B_0} x + \sum_{x \in B_1} x & = \sum_{i=0}^{n-1} x_i \\ \sum_{x \in B_0} x & \leq \lfloor \frac{\sum_{i=0}^{n-1} x_i}{2} \rfloor \\ \sum_{x \in B_1} x & \leq \lfloor \frac{\sum_{i=0}^{n-1} x_i}{2} \rfloor \end{cases} \Longrightarrow \sum_{x \in B_0} x = \sum_{x \in B_1} x$$

$\rightarrow$ OK

Donc PARTITION $\leq_p$ BPD

On a ainsi BPD $\in NP$ et PARTITION $\leq_p$ BPD (avec PARTITION NP-complet) ce qui permet de conclure.

2.2 Stratégies gloutonnes

$\boxed{5}$ Proposez un type *box* pour représenter une boîte (et son contenu). On souhaite pouvoir déterminer le volume disponible dans la boîte, et y ajouter un objet en temps constant.
Corrigé :

```
type box = {
  mutable capacity : int ;
  mutable content : int list}
```

$\boxed{6}$ Écrire une fonction `next_fit : instance -> box list` qui résout le problème en utilisant la stratégie *next-fit*.
Corrigé :

```
let next_fit (inst : instance) : box list =
  let c = fst(inst) in
  let elts = snd(inst) in

  let rec aux (current_box : box) (liste_elts : int list) : box list =
    match liste_elts with
    |[] -> [current_box]
    |x::[] ->
        if  x <= current_box.capacity then
          begin
            current_box.capacity <- current_box.capacity - x;
            current_box.content <- x::current_box.content;
            [current_box]
          end
        else let new_box = {capacity = c-x; content = [x]} in [current_box;new_box]
    |x::xs ->
        if  x <= current_box.capacity then
          begin
            current_box.capacity <- current_box.capacity - x;
            current_box.content <- x::current_box.content;
            aux current_box xs
          end
        else let new_box = {capacity = c-x; content = [x]} in current_box::(aux new_box xs)
  in
  let box_start = {capacity = c; content = []} in
  aux box_start elts
```

$\boxed{7}$ Écrire de même une fonction `first_fit : instance -> box list`
Corrigé :

```
let first_fit (inst : instance) : box list =
  let c = fst(inst) in
  let elts = snd(inst) in

  let rec aux_first_indice (liste_box : box list) (elt : int) : box list =
    match liste_box with
    |[] -> let new_box = {capacity = c-elt; content=[elt]} in [new_box]
    |x::xs when x.capacity >= elt -> (* Premiere fois qu'on tombe sur une case qu'on peut accueillir *)
        x.capacity <- x.capacity - elt ; x.content <- elt::(x.content); x::xs
    |x::xs -> x::(aux_first_indice xs elt)
  in

  let rec main_aux (liste_preservee : box list) (liste_elts : int list) : box list =
    match liste_elts with
    |[] -> liste_preservee
    |x::xs -> let updated_liste_preservee = aux_first_indice liste_preservee x in main_aux
        updated_liste_preservee xs
  in
  main_aux [] elts
```

$\boxed{8}$ Écrire de même une fonction `first_fit_decreasing : instance -> box list`

Corrigé :

```
let first_fit_decreasing (inst : instance) : box list =
  let c = fst(inst) in
  let elts = snd(inst) in
  let sorted_elts = List.sort compare elts in (* Utilisez votre algo de tri pref -> exo *)

  let rec aux_first_indice (liste_box : box list) (elt : int) : box list =
    match liste_box with
    |[] -> let new_box = {capacity = c-elt; content=[elt]} in [new_box]
    |x::xs when x.capacity >= elt -> (* Premiere fois qu'on tombe sur une case qu'on peut accueillir *)
        x.capacity <- x.capacity - elt ; x.content <- elt::(x.content); x::xs
    |x::xs -> x::(aux_first_indice xs elt)
  in

  let rec main_aux (liste_preservee : box list) (liste_elts : int list) : box list =
    match liste_elts with
    |[] -> liste_preservee
    |x::xs -> let updated_liste_preservee = aux_first_indice liste_preservee x in main_aux
        updated_liste_preservee xs
  in
  main_aux [] sorted_elts
```

$\boxed{9}$ Quelle solution obtient-on pour l'instance de la question 1 avec les différentes stratégies ?

Corrigé :

Pour $C = 10$ et $X = 2, 5, 4, 7, 1, 3, 8$ on a :

— *next-fit :*

$$B_0 = \{2;5\}$$
$$B_1 = \{4\}$$
$$B_2 = \{7;1\}$$
$$B_3 = \{3\}$$
$$B_4 = \{8\}$$

— *first-fit :*

$$B_0 = \{2;5;1\}$$
$$B_1 = \{4;3\}$$
$$B_2 = \{7\}$$
$$B_3 = \{8\}$$

— *first-fit-decreasing :*

$$B_0 = \{1;2;3;4\}$$
$$B_1 = \{5\}$$
$$B_2 = \{7\}$$
$$B_3 = \{8\}$$

$\boxed{10}$ Déterminer la complexité dans le pire cas des fonctions qui correspondent aux différentes stratégies.

Corrigé :

On a

— *next-fit :* $\mathcal{O}(n)$ avec $n = \#X$

— *first-fit :* $\mathcal{O}(n^2)$ (dans le cas où l'on doit ajouter à chaque fois, on parcourt donc la liste de longueur max n pour chaque elt).

— *first-fit-decreasing :* $\mathcal{O}(n^2 + \beta(n))$ avec $\beta(n)$ la complexité de votre algorithme de tri.

2.3 Analyse des approximations (cf. TD)

2.4 Analyse de *next-fit*

$\boxed{\textbf{12}}$ Montrer que $v_i + v_{i+1} > C$ pour $0 \leq i < m-1$.

Corrigé :

Supposons que $v_i + v_{i+1} \leq C$

Alors d'après le principe de la méthode *next-fit*, tous les objets de la boîte B_{i+1} sont en réalité dans la boîte b_i. Ce qui est donc absurde.

$\boxed{\textbf{13}}$ En déduire que *next-fit* fournit une 2-approximation pour BinPacking

Corrigé :

On a

$$V = \sum_{i=0, i \text{pair}}^{m-2} v_i + v_{i+1} > \frac{(m-1)}{2} C$$

or

$$V = \sum_{i=0}^{m^*-1} v_i \leq \sum_{i=0}^{m^*-1} C = m^* C \tag{1}$$

Or d'après la $\boxed{\textbf{12}}$,

$$\sum_{i=0, i \text{pair}}^{m-2} v_i + v_{i+1} = 2V - v_0 - v_{m-1} > (m-1)C$$

et étant donné que $v_0, v_{m-1} > 0$. on a bien

$$2V > (m-1)C$$

Ainsi (1) nous permet de conclure :

$$\begin{aligned} 2m^* C &\geq 2V > (m-1)C \\ \text{i.e } 2m^* &> m-1 \\ \text{i.e } m^* &\geq \frac{m}{2} \end{aligned}$$

D'où le résultat voulu

2.5 Analyse de *first-fit-decreasing*

On note m le nombre de boîtes utilisées par *first-fit-decreasing* et m^* le nombre optimal de boîtes. On note $x_0, \ldots, x_{n-1}$ les poids.

On considère la boîte B_j avec $j = \lfloor \frac{2m}{3} \rfloor$ et on note x l'objet de poids maximal rangé dans la boîte B_j.

Cas $x > \frac{C}{2}$

$\boxed{\textbf{14}}$ Montrer que $m \leq \frac{3}{2} m^*$

Corrigé :

On déduit plusieurs choses du fait que $x > \frac{C}{2}$, notons k_0 l'indice de x dans les $(x_k)_{0 \leq k \leq n-1}$:

1. Tous les poids d'indice $k < k_0$ dans rangés **individuellement** dans les boîtes d'indice $i < j$ parce qu'ils sont examinés avant x vu que la liste est triée par ordre décroissant.
2. On utilise donc **au moins** $j+1$ boîtes (car $j+1$ poids $> \frac{C}{2}$

On en déduit que toute solution au problème, en particulier la solution optimale, utilisera au moins $j+1$ boîtes. Or, par propriété de la partie entière :

$$j = \lfloor \frac{2m}{3} \rfloor \leq \frac{2m}{3} \leq j+1$$

Ainsi,

$$\begin{aligned} m^* &\geq j+1 \geq \frac{2}{3} m \\ \text{d'où } m^* &\geq \frac{2}{3} m \end{aligned}$$

D'où le résultat voulu.

Cas $x \leq \frac{C}{2}$

15 Montrer que $\sum_{l=j}^{m-1} v_l > v + 2v(m-j-1)$

Corrigé :
x étant de poids maximal dans B_j, c'est le premier élément à y être inséré et tous les éléments insérés dans les B_k avec $k \geq j$ sont donc considérés après, ils ont donc notamment tous un poids $\leq \frac{C}{2}$.
Ainsi, chaque boîte B_k (toujours avec $k \geq j$) peut contenir au moins deux éléments et en dehors de la dernière, elles vont effectivement contenir au moins deux éléments (sinon on n'ouvrirait pas la suivante).
De plus, ces éléments ne sont pas insérés dans les B_k avec $k < j$ donc ils ont chacun un poids $> v$.
Ainsi,

$$\forall i \in [|j; m-2|], v_i > 2v$$
$$\text{et } v_{m-1} > v$$

D'où le résultat

$$\boxed{\sum_{i=j}^{m-1} v_i > 2v(m-1-j) + v}$$

16 → à faire

2.6 Difficulté de l'approximation (cf. TD)

17 → à faire

3 N-reines : backtracking et Las Vegas

18 Écrire une fonction `bool check(int n, int sol[])`

Corrigé :

```c
bool check(int n, int sol[], int k){
    int colonne = sol[k];
    int ligne = k;

    // Check de la colonne
    for (int j=0; j<k; j++){
        if (sol[j]==colonne){
            // Si on trouve une reine sur la mm colonne
            return false;
        }
    }

    // Check de la diagonale droite
    int ligne_tmp = k-1;
    int colonne_tmp_droite = colonne+1;

    while(ligne_tmp >= 0 && colonne_tmp_droite <n){
        int colonne_tmp = sol[ligne_tmp];
        if (colonne_tmp == colonne_tmp_droite){
            // Si une reine se trouve sur la diag. droite
            return false;
        }
        ligne_tmp--;
        colonne_tmp_droite++;
    }

    // La colonne droite est ok, checkons la gauche
    int ligne_tmp_gauche = k-1;
    int colonne_tmp_gauche= colonne-1;
    while (ligne_tmp_gauche>= 0 && colonne_tmp_gauche>=0){
        int colonne_tmp = sol[ligne_tmp_gauche];
        if (colonne_tmp == colonne_tmp_gauche){
            // Si une reine se trouve sur la diag. gauche
            return false;
        }
        ligne_tmp_gauche--;
        colonne_tmp_gauche--;
    }

    // On a tout check, on peut renvoyer true
    return true;
}
```

19 Écrire une fonction `bool solve(int n, int sol[], int k)`

Corrigé :

```c
bool solve(int n, int sol[], int k){
    if (k==n){
        return true;
    }
    else{
        for (int colonne_k =0; colonne_k <n ; colonne_k++){
            sol[k]=colonne_k;
            if (check(n,sol,k)){
            // La position (k,colonne_k) est correcte, testons maintenant si
                if (solve(n,sol,k+1)){
                    return true;
                }
            }
            sol[k]=-1;
        }
        return false;
    }
}
```

20 Justifier que la procédure va choisir une colonne compatible de manière uniforme.

Corrigé :

On expose :

$$\text{Proba col\_choisie est la première colonne valide } = 1 \times \frac{1}{2} \times \ldots \times \frac{n-1}{n} = \frac{(n-1)!}{n!} = \frac{1}{n}$$
$$\text{Proba col\_choisie est la ième colonne valide } = \frac{1}{i} \times \ldots \times \frac{n-1}{n} = \frac{1}{n}$$

D'où le tirage uniforme

21 Écrire une fonction `bool solve_prob(int n, int sol[], int k)`

Corrigé :

```c
bool solve_prob(int n, int sol[], int k){
    if (k==n){
        return true;
    }
    else{
        int t = 0;
        int col_choisie = 0;

        for (int v=0; v<n ; v++){
            sol[k]=v;
            if (check(n,sol,k)){
                t++;
                if (rand() % t == 0){
                    col_choisie = v;
                }
            }
        }
        if (t==0){
            return false;
        }
        else{
            sol[k]= col_choisie;
            return solve_prob(n,sol,k+1);
        }
    }
}
```

22 Écrire une fonction de type LV.

Corrigé :

```c
void reinit_tab(int n, int tab[]){
    for (int i=0; i<n ; i++){
        tab[i]=-1;
    }
}

bool LV_solve_prob(int n, int sol[]){
    while (!solve_prob(n,sol,0)){
        reinit_tab(n,sol);
    }
    return true;
}
```