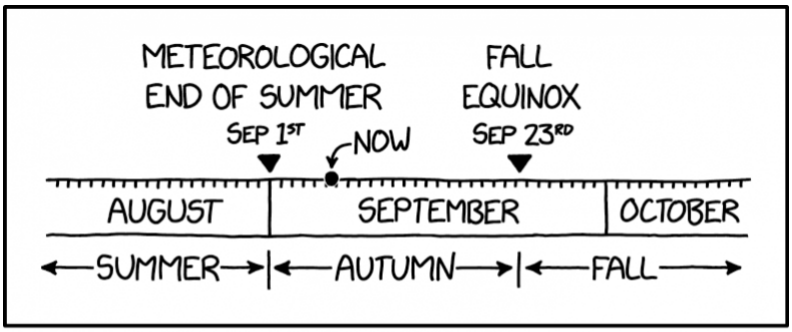
MPI* Info

Devoirs de vacances



NOW THAT SUMMER IS OVER, THE FIRST DAY OF FALL IS JUST A FEW WEEKS AWAY!

Olivier Caffier

Exercices classiques

Exercice 1 : Autour de la recherche dichotomique

Dans tout l'exercice, on ne considère que des tableaux d'entiers de longueur $n \geq 0$.

Un squelette de programme C vous est donné, avec un jeu de test qu'il **ne faut pas modifier**. Vous pouvez bien sûr ajouter vos propres tests à part.

1. Écrire une fonction `bool nb_occurences(int n, int* tab, int x)` qui renvoie le nombre d'occurences de l'élément `x` dans le tableau `tab` de longueur `n`. Quelle est la complexité de cette fonction ?
Corrigé :

```c
int nb_occurrences(int n, int* tab, int x) {
  int count = 0;
    for (int i =0; i<n; i++){
        if (tab[i]==x){
            count +=1;
        }
    }
  return count;
}
```

On parcourt tout le tableau une seule fois donc on se retrouve en $\mathcal{O}(n)$.

Dans toute la suite, **on suppose que les tableaux sont triés dans l'ordre croissant**.

On va chercher à écrire une version plus efficace de la fonction ci-dessus qui exploite cette propriété. On cherche tout d'abord à écrire une fonction `int une_occurence(int n, int* tab, int x)` qui permet de renvoyer un indice d'une occurence quelconque de l'élément `x` s'il est présent dans le tableau et `-1` sinon. On procède alors par dichotomie.

2. Compléter le code de la fonction `int une_occurence(int n, int* tab, int x)` qui renvoie l'indice de la première occurence d'un élément `x` dans un tableau `tab` de longueur `n`. Cette fonction devra avoir une complexité en $\mathcal{O}(\log n)$
Corrigé :

```c
int une_occurrence(int n, int* tab, int x) {
  int d = 0;
  int f = n - 1;
  int res = -1;

  while (d <= f) {
    int m = (d + f) / 2;
    if (tab[m]==x) {
      res = m;
      break;
    } else if (tab[m] < x) {
      d = m+1;
    }
    else {
      f = m-1;
    }
  }
  return res;
}
```

3. Écrire une fonction `int premiere_occurence(int n, int* tab, int x)` qui renvoie l'indice de la première occurence d'un élément `x` dans un tableau `tab` de longueur `n`. Cette fonction devra avoir une complexité en $\mathcal{O}(\log n)$.
Corrigé :

```c
int premiere_occurrence(int n, int* tab, int x) {
  int d = 0;
  int f = n - 1;
  int res = -1;

  while (d <= f){
    int m = (d+f)/2;
    if (tab[m]==x){
      res =m;
      f=m-1;
    }
    else if (tab[m] <x){
      d = m+1;
    }
    else{
      f = m-1;
    }
  }
  return res;

}
```

4. Écrire une fonction `int nombre_occurence(int n, int* tab, int x)` qui renvoie le nombre d'occurences de l'élément `x` dans le tableau `tab` de longueur `n`. Cette fonction devra avoir une complexité en $\mathcal{O}(\log n)$.
Corrigé :

```c
int nombre_occurrences(int n, int* tab, int x) {
  int d = 0;
  int f = n - 1;
  int count = 0;

  while (d <= f){
    int m = (d+f)/2;
    if (tab[m]==x){
      count+=1;
      int i_left = m-1;
      int i_right = m+1;
      while (i_left >= 0 && tab[i_left]==x){
        count+=1;
        i_left -=1;
      }
      while (i_right <= n-1 && tab[i_right]==x){
        count+=1;
        i_right +=1;
      }
      break;
    }
    else if (tab[m] <x){
      d = m+1;
    }
    else{
      f = m-1;
    }
  }
  return count;
}
```

5. Justifier que la fonction `une_occurence` termine et est correcte. On donnera un variant et un invariant de boucle que l'on justifiera.
Corrigé :
   - **Variant :** En notant f l'indice de fin et d l'indice de début dans notre intervalle de recherche, obtient l'invariant inv $= f - d$.
   - **Invariant :** $x \in$ `tab` $\Rightarrow x \in$`tab[d:f]`
     DÉMONSTRATION.
     On définit l'indice du milieu `m` et on fait face à trois cas :
     - Si `t[m] == x` : Bon, l'invariant ne sert plus trop parce qu'on a trouvé l'élément, mais on renvoie donc `true` ce qui est correct !
     - Si `t[m] > x` : Le tableau étant trié par ordre croissant, `x` ne peut que se situer dans l'intervalle $[d, m-1]$ (rappelons qu'il appartient au tableau `tab[d:f]` avec les anciennes valeurs de `d` et `f`). Ainsi, on va choisir les nouvelles valeurs de `d` et `f` tq. `d = d` et `f = m-1`. Comme dit précédemment, la présence de l'élément dans cet intervalle est garanti de par le caractère trié du tableau. On en déduit que $x \in$ `tab[d:f]`.
     - Si `t[m] < x` : De même, en prenant bien sûr `d = m + 1` et `f = f`.

     d'où le résultat voulu.

6. Montrer que la complexité de la fonction `une_occurence` est bien en $\mathcal{O}(\log n)$.
Corrigé :
Notons $C(n)$, pour une entrée de taille $n \in \mathbb{N}$, le nombre d'opérations effectuées par l'algorithme dans le pire des cas. Ainsi, dans le pire des cas, on effectue une comparaison et on continue la recherche sur une entrée de taille divisée par 2, d'où :

$$C(n) = C(\frac{n}{2}) + 1$$

Intéressons-nous aux entrées dont les tailles sont des puissances de 2. On a alors, pour $k \in \mathbb{N}$, :

$$C(2^{k+1}) = C(2^k) + 1$$

On retrouve alors une suite arithmétique de raison 1, d'où :

$$C(2^k) = k$$

Ainsi, en prenant $k = \log n$, on a bien :

$$C(n) = \log(n)$$

d'où la complexité $\mathcal{O}(\log n)$.

Exercice 2 : Plus court chemin dans un DAG

Nous allons ici écrire un programme qui permet de calculer les plus courts chemins dans un graphe orienté acyclique pondéré $G = (S, A, p)$ (on considère les sommets indexés de 0 à $|S|-1$) et un sommet `s` $\in S$, on veut renvoyer un tableau `d` de taille $|S|$ tel que pour tout `i` $\in S$, `d[i]` contienne le poids minimal d'un chemin de `s` à `i`.

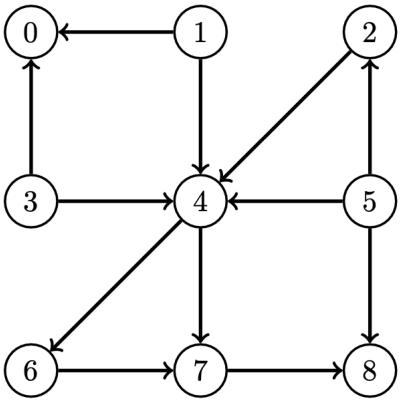
On utilisera la structure suivante pour représenter les graphes pondérés :

```
struct graph_s {
  int n;
  int degre [100];
  int voisins [100] [10];
  int poids [100][10];
};
```

L'entier `n` correspond au nombre de sommet $|S|$ du graphe. On suppose que `n` ≤ 100. Pour $0 \leq$ `s` $\leq$ `n`, la case `degre[s]` contient le degré d^+(`s`), c'est-à-dire le nombre de successeurs, appelés ici voisins, de `s`. On suppose que ce degré est toujours inférieur à 10. Pour $0 \leq$ `s` $<$ `n`, la case `voisins[s]` est un tableau contenant, aux indices $0 \leq$ `i` $< d^+$(`s`), les voisins du sommet `s` et pour $0 \leq$ `s` $<$ `n`, la case `poids[s]` est un tableau contenant, aux indices $0 \leq$ `i` $< d^+$(`s`), les poids des arcs `(s,voisins[s][i])`.

Il s'agit donc d'une représentation par liste d'adjacence où les listes sont représentées par des tableaux en C.

Un programme en C vous est fourni dans lequel le graphe ci-dessous est représenté par la variable `g_exemple`.



1. Avec cette représentation, quel est le nombre maximum de sommets que le graphe peut avoir ? Le nombre d'arcs ?
   Corrigé :
   On peut avoir au maximum 100 sommets, qui peuvent être reliés à 10 autres (eux inclus), donc 1000 arcs au maximum.

2. Écrire une fonction `void mise_a_jour(struct graph_s g, int u, int v, int* d)` telle que l'appel à `mise_a_jour(g,u,v,d)` met à jour la case `d[g.voisins[u][v]]` avec le minimum entre sa valeur actuelle et la valeur de `d[u] + p(u,g.voisins[u][v])`.
   Corrigé :

```
void mise_a_jour(struct graph_s g, int u, int v, int* d){
  int i=0;

  while(g.voisins[u][i]!=v){
    i+=1;
  }

  if(g.poids[u][i]+d[u]<d[v]){
    d[v]=g.poids[u][i]+d[u];
  }
}
```

3. Écrire une fonction `int* creer_tableau_distances(struct graph_s g, int s)` qui alloue et initialise un tableau des distances `d` de bonne taille tel que `d[s]=0` et pour tous les autres sommets `x`, `d[x]=10000`.
Corrigé :

```
int* creer_tableau_distances(struct graph_s g, int s){
    int* d = malloc(g.n*sizeof(int));
    for(int i=0;i<g.n;i++){
      d[i] = 10000;
    }
    d[s] = 0;
    return d;

}
```

Le principe de l'algorithme de recherche de plus court chemin est le suivant : on initialise un tableau `d` avec la fonction précédente et on trie par ordre topologique les sommets du graphe puis on parcourt ces derniers dans cet ordre et pour chaque sommet `u`, on appelle la fonction `mise_a_jour(g,u,v,d)` pour chaque arc d'origine `u`.

4. Compléter la fonction `tri_topologique` fournie qui permet d'obtenir un ordre topologique.
Corrigé :

```
void tri_topologique(graph g, bool vu[100],int liste[100]){
  bool* vu_parc = malloc(100*sizeof(bool));
  for(int i=0;i<100;i++){
    vu_parc[i]=false;
  }
    for (int i = 0; i<g.n;i++){
        if (!vu[i]){
          vu[i] = true;
          int indice = g.n-1;
          for(int j = 0;j<g.n;j++){
            parcours_prof_rec(g,j,vu_parc,liste,&indice);
          }

        }
    }
    free(vu_parc);
}
```

5. Écrire une fonction `int* plus_court_chemin(struct graph_s g, int s)` qui renvoie le tableau des poids minimaux des chemins `s` et chacun des sommets du graphe à l'aide de l'algorithme proposé.
Corrigé :

```
int* plus_court_chemin(graph g, int s){
  int* d=creer_tableau_distances(g,s);
  int* liste = malloc(g.n*sizeof(int));
  bool* vus = malloc(g.n*sizeof(bool));
  for(int i=0;i<g.n;i++){
    liste[i]=i;
    vus[i]=false;
  }
  tri_topologique(g,vus,liste);
  int i =0;
  while(liste[i]!=s){
    i+=1;
  }
  for(int j =i;j<g.n;j++){
    for(int k =0;k<g.degre[liste[j]];k++){

      mise_a_jour(g,liste[j],g.voisins[liste[j]][k],d);
    }
  }
  free(liste);
  free(vus);
  return d;
}
```

6. Prouver la correction de l'algorithme proposé calculant les plus courts chemins dans un graphe orienté pondéré acyclique.
   Corrigé : Soit $u_0, \dots, u_n = t$ un plus court chemin. On montre par rec. sur $k \in [\![0]; n]\!]$ que lorsque l'on traite les voisins de u_k, alors $d(u_k) = \delta(s, u_k)$.
7. Écrire une fonction `bool detect_cycle(struct graph_s g)` qui vérifie si un graphe orienté pondéré est bien acyclique.
   Corrigé :

```c
bool detect_cycle_sub(graph g,bool* vus,int s){
  if(!vus[s]){
    vus[s]=true;
    for(int j= 0;j<g.degre[s];j++){
      detect_cycle_sub(g,vus,g.voisins[s][j]);
    }
    return false;
  }
  else{
    return true;
  }
}

bool detect_cycle(graph g){
  bool* vus = malloc(g.n*sizeof(bool));
  for(int i=0;i<g.n;i++ ){
    vus[i]=false;
  }
  bool res = detect_cycle_sub(g,vus,0);
  free(vus);
  return res;
}
```

Exercice 3 : Timsort

On remarque qu'en pratique, les tris par comparaisons ont rarement des entrées trop désordonnées. On sait par ailleurs que le tri rapide a de bonnes performances sur des tableaux bien désordonnés, ce qui ne correspond donc pas forcément à la réalité. C'est pourquoi des modification récentes des librairies utilisent des algorithmes de tri qui sont performants sur des entrées dites "pseudo-triées".

Toute liste l non vide peut être décomposée en $\rho \geq 1$ séquences croissantes maximales, qui sont des sous listes croissantes $l_1, \ldots, l_p$ dont la concaténation fait l et telles que pour tout $i \in [\![1; \rho - 1]\!]$, le dernier élément de l_i est strictement plus grand que le premier élément de l_{i+1}.

On définit la SCM d'une liste l comme la liste de listes de taille ρ qui contient, dans l'ordre, les séquences définies précédemment.

Par exemple, si on considère la liste

`[2;4;20;29;3;3;4;1;1;4;32;8;11]`

Alors sa SCM est représentée par

`[[2;4;20;29];[3;3;4];[1;1;4;32];[8;11]]`

1. Écrire une fonction `scm : int list -> int list list` telle que `scm l` renvoie la SCM de l'entrée `l`. On attend une complexité linéaire en la taille de la liste passée en entrée.
   Corrigé :

```
let reverse_full (l : int list list) : int list list =
  let tmp_res = List.rev l in
  let rec aux (li : int list list) : int list list =
    match li with
    |[] -> []
    |x::xs -> (List.rev x)::(aux xs)
  in aux tmp_res

let scm (l : int list) : int list list =
  let rec aux (l_main : int list list) (l1 : int list) (l2 : int list) : int list list =
    match l1,l2 with
    |_,[] -> l1::l_main
    |[],x::xs -> aux l_main [x] xs
    |y1::x1, x2::[] when y1 > x2 -> let tmp = l1::l_main in l2::tmp
    |y1::x1, x2::x2s when x2 >= y1 -> aux l_main (x2::l1) x2s
    |y1::x1, x2::x2s -> aux (l1::l_main) ([x2]) x2s
  in reverse_full (aux [] [] l)
```

2. Écrire une fonction `fusion : int list -> int list -> int list` telle que si on considère deux listes triées `l1` et `l2` , alors `fusion l1 l2` renvoie une liste triée qui contient les éléments de `l1` et ceux de `l2`. Quelle est la complexité (nombre de comparaisons) de votre fonction dans le pire cas ?
   Corrigé :

```
let rec fusion (l1 : int list) (l2 : int list) : int list =
  match l1,l2 with
  |_,[] -> l1
  |[],_ -> l2
  |x::xs,y::ys when x > y -> y::(fusion l1 ys)
  |x::xs,y::ys -> x::(fusion xs l2)
```

On est en $\mathcal{O}(|l_1| + |l_2|)$.

L'objectif ici est, à partir de la SCM d'une liste `l`, de trouver une succession de fusions à effectuer sur les séquences la composant pour obtenir une liste triée qui contient les éléments de `l`. Une représentation d'une succession de fusions peut être faite par un arbre binaire strict dont les feuilles sont les listes et un noeud interne correspond à la fusion des deux listes triées obtenues par la représentation de chacun de ses deux fils. L'arbre choisi aura une influence sur la complexité finale obtenue.

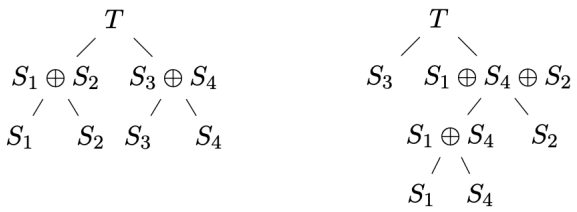
Par exemple, considérons de nouveau la liste

`[2;4;20;29;3;3;4;1;1;4;32;8;11]`

avec sa SCM

`[[2;4;20;29];[3;3;4];[1;1;4;32];[8;11]]`

Si on note S_1 = `[2;4;20;29]`, S_2 = `[3;3;4]`, S_3 = `[1;1;4;32]` et S_4 = `[8;11]`, on peut obtenir l'un des deux arbres suivant pour représenter la succession de fusions :



3. Dans chacun des deux cas, donner le nombre de comparaisons total de l'ensemble des fusions effectuées sur l'exemple.
Corrigé :
   - 1er cas : $4+5+12=21$ comparaisons.
   - 2ème cas : $4+4+12=20$ comparaisons.

   On utilisera le type suivant pour décrire un arbre binaire strict :

```
type arbre = Feuille of list | Noeud of (arbre*arbre)
```

   Les arbres de la figure ci-dessous correspondent donc aux arbres suivants :

```
let s1 = [2;4;20;29];;
let s2 =[3;3;4];;
let s3 =[1;1;4;32];;
let s4 = [8;11];;

let a1 = Noeud(Noeud(Feuille(s1),Feuille(s2)),Noeud(Feuille(s3),Feuille(s4)));;
let a2 = Noeud(Feuille(s3),Noeud(Noeud(Feuille(s1),Feuille(s4)),Feuille(s2)));;
```

4. Écrire une fonction `sort_tree : arbre -> list` qui, à partir de l'arbre des fusions, renvoie la liste triée.
Corrigé :

```
let rec sort_tree (t : arbre) : int list =
  match t with
  |Feuille(l) -> l
  |Noeud(sa1,sa2) -> fusion (sort_tree sa1) (sort_tree sa2)
```

5. Une solution est de construire un arbre le plus équilibré possible (arbre binaire strict complet, c'est-à-dire tel que toutes les feuilles sont à la même hauteur ou ont une différence de hauteur d'au plus un).
Écrire une fonction `constr_arbre : int list list -> arbre` qui construit un tel arbre des fusions.
Quelle est la complexité de votre construction?
Corrigé :

```
let constr_arbre (l : int list list) : arbre =
  let tmp_array = Array.of_list l in
  let len = Array.length tmp_array in
  let rec aux (tab : int list array) (start : int) (ending : int) : arbre =
    match start,ending with
    |start, ending when ending < start -> failwith "NON"
    |start,ending when ending = start -> Feuille(tab.(ending))
    |start, ending when ending-start = 1 -> Noeud( Feuille(tab.(start)), Feuille(tab.(ending)))
    |_,_ -> let mid = (start + ending) / 2 in
        let left =  aux tab start mid in
        let right = aux tab (mid+1) ending in
        Noeud(left, right)
  in aux tmp_array 0 (len-1)
```

6. Écrire une fonction `tri : int list -> int list` qui trie une liste passée en entrée suivant l'algorithme proposé. En déduire une fonction de tri.
   Corrigé :

```
let tri (l : int list) : int list =
  let sous_seq = scm l in
  let abr = constr_arbre sous_seq in
  sort_tree abr
```

7. Déterminer le coût de votre algorithme en fonction de la taille `n` de la liste et de ρ, le nombre de SCM composant la liste.
8. Revenons au cas général, où l'arbre des fusions peut être quelconque : donner une formule qui décrit la complexité de l'algorithme de tri en fonction des profondeurs des différents noeuds de l'arbre et de la taille des séquences de la SCM. À l'aide d'un algorithme au programme, commenter la complexité d'un algorithme glouton qui permettrait de construire un arbre des fusions optimal. Quelle est sa complexité de construction ?

Exercice 4 : Algorithme de Wigderson

On dit qu'un graphe $G = (S, A)$ admet une coloration $c : S \to [\![0; k-1]\!]$ avec k couleurs ssi $\forall (x, y) \in A, c(x) \neq c(y)$.

Pour trouver une coloration d'un graphe, on peut utiliser l'algorithme glouton suivant :

> Pour chaque sommet $s \in S$, on lui attribue la plus petite couleur non utilisée par un de ses voisins déjà colorié.

Commençons par coder cet algorithme :

Un graphe sera représenté ici par sa matrice d'adjacence (donc un `bool array array`), ainsi un graphe à n sommets sera représenté par un tableau de dimension 2 de taille $n \times n$. Une coloration sera représentée par un tableau d'entier (un `int array`) de taille n, noté `c` tel que la case `c.(i)` donne la couleur du sommet `i`. On attribuera la valeur -1 à un sommet qui n'est pas encore colorié.

1. Écrire une fonction `couleur_a_utiliser : bool array array -> int array -> int -> int -> int` telle que `couleur_a_utiliser g coloration s c` renvoie la plus petite couleur supérieure ou égale à `c` non déjà utilisée par un voisin de `s` dans `g` partiellement colorié avec coloration.
   Corrigé :

```
let couleur_a_utiliser (g : bool array array) (coloration : int array) (s : int) (c : int) : int =
  let len = Array.length g in
  let couleur_utilisee = Array.make len false in
  for i = 0 to len -1 do
    if g.(s).(i) && coloration.(i) <> -1 then
      couleur_utilisee.(coloration.(i)) <- true
  done;
  let count = ref c in
  let res = ref len in
  while !count <= len-1 do
    if !count<= !res && (not couleur_utilisee.(!count)) then begin res := !count ; incr(count) end
    else incr(count);
  done;
  !res
```

2. En déduire une fonction `coloriage : bool array array -> int array` qui renvoie une coloration du graphe passé en entrée en utilisant l'algo glouton.
   Corrigé :

```
let coloriage (g : bool array array) : int array =
  let len = Array.length g in
  let color = Array.make len (-1) in
  for i = 0 to len-1 do
    let tmp = couleur_a_utiliser g color i 0 in
    color.(i)<- tmp
  done;
  color
```

3. Donner un majorant du nombre de couleurs utilisées par cet algorithme.
   Corrigé : En notant $\tilde{c}$ le nombre de couleurs utilisées par cet algorithme, on a :

$$\boxed{\tilde{c} \leq d}$$

où d est le degré maximal du graphe.

L'algorithme de Wigderson permet de trouver un coloriage utilisant $\mathcal{O}(\sqrt{n})$ couleurs pour un graphe dont on sait qu'il est 3-coloriable.

Cet algorithme fonctionne selon le schéma suivant :

- On fixe initialement c à 0.
- Pour chaque sommet $s \in S$ ayant au moins $\sqrt{(n)}$ voisins non encore coloriés :
  - On 2-colorie avec les couleurs $c+1$ le sous graphe induit par les voisins de s non encore coloriés.
  - On incrémente c du nombre de couleurs utilisées lors de cette étape.
  - On utilise l'algorithme glouton pour colorier avec des couleurs supérieures ou égales à c les sommets non coloriés.

Un graphe induit $G' = (S', A')$ du graphe G sera représenté par une matrice d'adjacence `g_prime` de même taulle que celle de G où la case `g_prime.(i).(j)` contient `1` ssi $i, j \in S'$ et $(i, j) \in A$. La connaissance de cette matrice ne suffisant pas à connaître le graphe induit, on utilisera (si besoin) en plus un tableau de booléens de taille $|S|$ dont la case d'indice `i` contiendra `true` ssi $i \in S'$.

4. Écrire une fonction `deux_color : int array array -> int ->int array-> bool array->unit` qui prend en entrée la matrice d'adjacence d'un sous graphe induit de `g`, une couleur `c`, un coloriage partiel de `g` et un tableau de booléens qui correspond à l'ensemble des sommets effectivement dans le graphe induit et qui met à jour le coloriage en 2 coloriant le graphe induit avec les couleurs `c` et `c + 1`.
Corrigé :

```
let deux_color (g : int array array) (c : int) (coloration : int array)(tab : bool array) : unit =
  let len = Array.length g in
  for i = 0 to len-1 do
    let tmp = ref c in
    if tab.(i) && coloration.(i) = -1 then
      begin
        for j = 0 to len-1 do
          if g.(i).(j)=1 && coloration.(j) = c then tmp := c+1
        done;
        coloration.(i) <- !tmp
      end
  done
```

5. Justifier que si le graphe est 3-coloriable, alors l'étape 1 est toujours possible, c'est-à-dire que les sous-graphes considérés 2-coloriables.
Corrigé :
Soit donc un graphe $G = (S, A)$ qu'on suppose 3-coloriable. Prenons un sommet $s \in S$ quelconque. Supposons (par l'absurde) que le sous-graphe induit à partir de ce sommet ne soit pas 2-coloriable : alors, en rajoutant s dans ce sous-graphe, on en déduit qu'il ne peut pas être 3-coloriable, ce qui est donc absurde.

6. Écrire une fonction `graphe_restant : bool array array -> int array-> int -> bool array array` telle que `graphe_restant g coloration s` renvoie la matrice d'adjacence du graphe induit par l'ensemble des voisins non coloriés de `s` dans `g` partiellement colorié par coloration.
Corrigé :

```
let graphe_restant (g : bool array array) (coloration : int array) (s : int) : bool array array =
  let len = Array.length g in
  let res = Array.make_matrix len len false in
  let parcours = Array.make len false in
  for i = 0 to len-1 do
    if  g.(s).(i) && coloration.(i) = -1 then parcours.(i)<-true
  done;
  for i = 0 to len-1 do
    if parcours.(i) then
      for j=0 to len-1 do
        if parcours.(j) && g.(i).(j) then res.(i).(j) <- true
      done;
  done;
  res
```

7. Écrire une fonction `graphe_glouton : bool array array -> int array -> bool array array` qui prend en entrée un graphe et une coloration partielle de celui-ci et qui renvoie la matrice d'adjacence du graphe induit par l'ensemble des sommets non coloriés.
Corrigé :

```
let graphe_glouton (g : bool array array) (coloration : int array) : bool array array =
  let len = Array.length coloration in
  let res = g in
  for i = 0 to len-1 do
    if coloration.(i) = -1 then
      for j = 0 to len-1 do
        (* On considere que le graphe est oriente *)
        match g.(i).(j), g.(j).(i) with
        |true,true -> g.(i).(j) <- false;g.(j).(i) <- false
        |_,true -> g.(j).(i) <- false
        |true,_ -> g.(i).(j) <- false
        |_,_ -> ()
      done
  done;
  res
```

8. Écrire une fonction `continuer : bool array array -> int array -> int option` telle que `continuer g coloriage` calcule le nombre maximal de voisins non déjà coloriés d'un sommet. On note ce degré maximale `d` et la fonction renvoie `Some s` si $d \geq \sqrt{n}$ et que le nombre de voisins non coloriés de `s` est `d` et `None` sinon.
Corrigé :

```
let continuer (g : bool array array) (coloriage : int array) : int option =
  let len = Array.length g in
  let i = ref 0 in
  let res = ref None in
  while !i <= len-1 do
    let count = ref 0 in
    for j = 0 to len-1 do
      if g.(!i).(j) && coloriage.(j) = -1 then incr(count)
    done;
    if float_of_int(!count) >= sqrt(float_of_int(len)) then begin res := Some !i; i := len end
    else incr(i)
  done;
  !res
```

9. Écrire une fonction `wigderson : bool array array -> int array` qui renvoie un coloriage du graphe passé en entrée obtenu à l'aide de l'algorithme de Wigderson.
Corrigé :

```
let turn_into_int_array (g : bool array array) (sous_g : bool array array) (coloration : int array) (s
    : int) : int array array * bool array =
  let len = Array.length g in
  let res1 = Array.make_matrix len len 0 in
  let res2 = Array.make len false in

  (* PREMIER PARCOURS *)
  for i = 0 to len-1 do
    if g.(s).(i) && coloration.(i) = -1 then
      (*Le sommet i fait partie du sous_graphe induit car il est relie a s et n'est pas colorie *)
      res2.(i)<-true
  done;

  (* DEUXIEME PARCOURS *)
  for i = 0 to len-1 do
    if res2.(i) then
      for j =0 to len-1 do
        if res2.(j) && g.(i).(j) then
          (* Condition de l'enonce*)
          res1.(i).(j)<-1
      done
  done;
  res1,res2

```

```
let wigderson (g : bool array array) : int array =
  let len = Array.length g in
  let c = ref 0 in
  let color = Array.make len (-1) in
  let index = ref (continuer g color) in
  while !index <> None do
    match !index with
    |None -> failwith "Cas impossible"
    |Some s -> let sous_g = graphe_restant g color s in
        let sous_g_transfo,tab = turn_into_int_array g sous_g color s in
        deux_color sous_g_transfo !c color tab; c := !c +2; index := continuer g color
  done;
  let final_g = graphe_glouton g color in
  let final_color = coloriage final_g in
  for i = 0 to len-1 do
    if color.(i) = -1 then color.(i)<- final_color.(i) + !c
  done;
  color
```

Navré pour ces fonctions auxiliaires, c'est juste insupportable...

10. Montrer que cet algorithme utilise $\mathcal{O}(\sqrt{n})$ couleurs.

Corrigé :

On rappelle les deux étapes de l'algorithme :

(a) Pour tout $s \in S$ non déjà colorié et avec au moins $\sqrt{n}$ voisins non coloriés : on 2-colorie le sous-graphe induit avec les couleurs $c, c+1$. On ajoute à c le nombre de couleurs utilisées.

(b) On finit de colorier avec l'algo glouton en utilisant des couleurs plus grandes que c.

Donc,

- Soit donc m le nombre de sommets traités durant l'étape (a). On aura donc au moins $m\sqrt{n}$ sommets coloriés. Donc $m\sqrt{n} \leq n$, d'où (en divisant de manière malicieuse) :

$$m \leq \sqrt{n}$$

  De plus, à chaque fois on 2-colorie DONC le nombre de couleurs utilisées dans la première étape est d'au plus $2\sqrt{n}$.
- Ensuite, on considère le graphe restant utilisé pour la deuxième étape qui comportera au plus $\sqrt{n}-1$ sommets non coloriés entre eux. L'algo utilisera donc au plus $\sqrt{n}$ couleurs.
- Finalement (et pas « Au final »), on utilisera au plus $3\sqrt{n}$ couleurs.

D'où le $\mathcal{O}(\sqrt{n})$.

Planche CCINP 5

Partie A

Après un changement de l'équipe dirigeante, l'entreprise Mondelez International, qui fabrique les barres Toblerone, décide de rationaliser sa production pour maximiser ses revenus. En effet, leur chaîne de production fabrique des barres de n "carreaux" qui sont ensuite coupées en barres plus petites avant d'être vendues. Mais une récente étude de marché a déterminé le prix auquel on pouvait vendre des barres de longueur k (pour $1 \leq k \leq n$), et ce prix s'avère ne pas avoir de relation simple avec k. Le problème est donc de décider comment découper la barre initiale de n carreaux en des barres plus petites pour maximiser le prix de vente total. Pour simplifier, on néglige le coût de la découpe et de l'emballage.

Dans tout le problème, on considérera que l'on dispose d'un tableau `t`, indicé de 0 à n, tel que `t[k]` soit le prix de vente d'une barre de longueur `k`, et que le prix d'un morceau de taille 0 est 0 (autrement dit que `t[0] = 0`). Remarquez que si `t[n]` est suffisamment grand, la solution optimale peut très bien être de ne pas découper la barre.

On donne un exemple de tableau `t` pour $n = 10$:

```
int t[10] = { 0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 26 };
```

1. Identifier le découpage en trois blocs qui permet d'obtenir un prix de vente optimal sur l'exemple.
   Corrigé :
   On prend 2 tranches de longueur 2 et une de longueur 6 et on obtient un prix de vente optimal de 27 € (modulo standard monétaire de l'exercice). On vérifiera ce résultat plus tard.

2. On fixe $n \in \mathbb{N}$. Une découpe d'une barre de taille n en k morceaux est la donnée d'un k-uplet $(c_1, \ldots, c_k)$ tel qu'on aura une barre de taille c_i pour chaque $1 \leq i \leq k$. Avec cette notation, que vaut $\sum_{i=1}^{k} c_i$?
   Corrigé : On a
   $$\boxed{\sum_{i=1}^{k} c_i = n}$$

3. Si on voulait calculer le prix de chaque découpe afin d'en déterminer la valeur maximale, combien de cas devrait-on considérer ?
   Corrigé :
   On dispose à chaque fois le choix de découper ou non le "carreau" (à part pour le dernier). Ces choix étant totalement indépendants, on peut dire qu'à chaque fois on a deux possibilités et donc par principe multiplicatif, on a
   $$\boxed{2^{n-1} \text{ découpes possibles.}}$$
   Bon, en fait il y a des situations assez analogues donc on en calculera un peu moins avec un algorithme de type brute-force, mais **on reste sur une complexité exponentielle** !

4. Pour arriver à une solution efficace, la première étape est de remarquer qu'une solution du problème (i.e une découpe optimale d'une barre de taille n) "contient" des solutions à des versions plus petites du même problème. Quel paradigme de programmation semble ici adapté .
   Corrigé :
   On pourrait penser à une stratégie "divide and conquer" en premier lieu mais cela ne permettrait pas de déterminer la découpe optimale (même si découper c'est diviser en soit). En fait, diviser pour régner est une stratégie qui fonctionne quand l'on peut réellement séparer (en parts égales ou non) notre problème. Ici, rien ne garantit qu'en subdivisant notre problème d'une certaine façon, on aura une découpe optimale ! C'est d'ailleurs pour cela que dans notre relation de récurrence, on dépendra de plusieurs subdivisions et on prendra le maximum de tous ces résultats. **Il faut donc s'intéresser à la programmation dynamique !**

5. On peut supposer (sans perte de généralité), qu'à chaque étape d'une solution optimale, on découpe ce qui reste de la barre en un bloc de taille k (avec $1 \leq k \leq n$) qui fera partie de la découpe finale et une nouvelle barre qui pourra à nouveau être découpée. Justifier que cette nouvelle barre (de longueur $n-k$) devra elle aussi être découpée de manière optimale.
Corrigé :
Si elle n'était pas découpée de manière optimale, alors on en déduit qu'il existe un autre découpage optimal $\tilde{c}_{n-k}$ tel que (c_k, c_{n-k}) soit optimal. Alors certes, il n'y a pas unicité du découpage optimal mais en tout cas, cela veut surtout dire que la découpe que l'on va renvoyer, qui est censée être optimale, ne l'est pas car on en a trouvé une meilleure. Ce qui est absurde car on construit une solution optimale.
6. On note $v(n)$ le prix de vente optimal pour une barre de n "carreaux" pour $n \in \mathbb{N}$. Que vaut $v(0)$? Donner une relation de récurrence pour $v(n)$.
Corrigé :
On a :
$$\begin{cases} v(0) = 0 \\ \forall n \in \mathbb{N}^*, v(n) = \max\limits_{k \in [\![0;n]\!]} t(k) + v(n-k) \end{cases}$$
avec $t(k)$ le prixde vente d'une tranche de longueur k.
7. Proposer un algorithme qui calcule $v(n)$ sur l'entrée n.
Corrigé :

```
let calcul (n_cases : int) (price : int array) : int =
  let v = Array.make (n_cases+1) (-1) in
  v.(0) <- 0;
  let rec aux (n : int) : int =
    if v.(n) = -1 then begin
      let max_v = ref (price.(n)) in
      for k = 1 to n do
        let v_nk = aux (n-k) in
        if price.(k) + v_nk > !max_v then max_v := price.(k) + v_nk
      done;
      v.(n) <- !max_v ;
      v.(n)
    end
    else v.(n)
  in aux n_cases
```

8. Quelle est la complexité de votre algorithme ?
Corrigé :
On a un algorithme récursif où l'on peut dire qu'avec un appel sur une entrée de taille n, on fait une boucle qui va de 1 à n et qui effectue à chaque fois un appel récursif. Remarque : on ne calculera qu'une seule fois (grâce à la mémoïsation) les $(v(k))_{k \in [\![0;n]\!]}$.
On dispose donc d'une complexité quadratique : $\mathcal{O}(n^2)$.

Partie B : En C

Dans cet exercice, on considère des arbres binaires non étiquetés. On dit qu'un arbre est binaire strict si tout nœud interne a exactement deux enfants, ou autrement dit si tout nœud (interne ou non) a zéro ou deux enfants. On dit qu'un arbre est parfait s'il est binaire strict et que toutes ses feuilles sont à même profondeur. On dit qu'il est presque-parfait si tous les niveaux de profondeur sont remplis par des nœuds, sauf éventuellement le dernier, rempli par la gauche.

1. Combien existe-t-il d'arbres presque-parfaits de taille 6? Représentez-les graphiquement.
   Corrigé :
2. Citer une structure de données qui peut s'implémenter avec des arbres presque-parfaits.
3. Proposer une définition par induction d'un arbre parfait de hauteur n, pour $n \in \mathbb{N}$. Justifier rigoureusement que cette définition coïncide avec celle de l'énoncé.

On implémente un arbre binaire en C par le type arbre défini de la manière suivante :

```c
struct Noeud {
  struct Noeud* gauche;
  struct Noeud* droite;
};
typedef struct Noeud arbre;
```

4. Écrire une fonction `int hauteur(arbre* a)` qui prend en argument un pointeur vers un arbre `a` et renvoie sa hauteur.
5. En déduire une fonction `bool est_parfait(arbre* a)` qui prend en argument un pointeur vers un arbre `a` et renvoie un booléen qui vaut `true` ssi l'arbre pointé est parfait.
6. Écrire une fonction `arbre* plus_grand_presque_parfait(arbre* a)` qui prend en argument un pointeur vers un arbre `a` et renvoie un pointeur vers le plus grand sous-arbre de `a` qui est presque parfait. La complexité de cette fonction doit être linéaire en la taille de l'arbre passé en entrée.

Indication : que peut-on dire des enfants gauche et droit d'un arbre presque-parfait de hauteur h?