Devoir Surveillé 5

15 mars 2025

INFORMATIQUE MP2I

DURÉE DE L'ÉPREUVE : **4 heures**

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Ce sujet comporte huit pages numérotées de 1/14 à 14/14

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 4 parties indépendantes, utilisant les langage de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I s'intéresse à la règle de Galil qui permet d'améliorer la complexité de l'algorithme de Boyer-Moore dans le pire des cas.
- La partie II s'intéresse au problème de la recherche dans une matrice dont les lignes et les colonnes sont triées et propose un algorithme plus efficace que de couper la matrice en 4 parts égales.
- La partie III s'intéresse à la recherche du couple de points le plus proches dans un nuage en points à l'aide d'une droite de balayage et illustre les différents avantages et inconvénients des listes par rapport aux tableaux.
- La partie IV s'intéresse au problème des huit dames.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes <assert.h>, <stdbool.h>, etc. doivent être inclus. En OCaml, les noms des arguments des fonctions sont donnés avec des indications de type. Il n'est pas nécessaire de recopier ces indications de type sur votre copie.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1 Boyer-Moore : the Galil rule – C – (40 pts)

Dans cette partie, nous nous intéressons à l'algorithme de Boyer-More présenté en classe, dont voici une version en C :

Boyer-More algorithm **C**

```
int search(char pattern[], char text[]){
  // Affiche les indices auxquels on trouve le motif et renvoie le nombre d'occurences
  int cpt = 0;
  int n = strlen(text);
  int m = strlen(pattern);

  int **table = bad_char(pattern);

  for (int i = 0; i <= n - m;){
    for (int j = m - 1; j >= 0; j = j - 1){
      unsigned char c = text[i+j];
      if (c != pattern[j])
        { i = i + table[j][c]; break; }
      if (j == 0){
        cpt = cpt + 1;
        printf("Occurence en position %d\n", i);
        i = i + 1;
      }
    }
  }

  free_bad_char(table, m);
  return cpt;
}
```

☐ 1 – Donner la table de décalage pour le mot *cancre*, pour la règle "bad character".

Réponse 1.

	c	a	n	r	e	_
0	x	1	1	1	1	1
1	1	x	2	2	2	2
2	2	1	x	3	3	3
3	x	2	1	4	4	4
4	1	3	2	x	5	5
5	2	4	3	1	x	6

On souhaite à présent améliorer cet algorithme lorsqu'il trouve une occurence. On peut pour cela utiliser les répétitions dans le motif que l'on cherche.

☐ 2 – Justifier brièvement que la relation d'ordre $\preceq$ définie sur Σ^* par $u \preceq v$ si et seulement si $u = v$ ou $|u| < |v|$ est une relation d'ordre bien fondée et non totale (il n'est pas nécessaire de montrer qu'elle est réflexive, transitive et anti-symétrique).

Réponse 2. Supposons qu'il existe une suite infinie $(u_n)_{n\in\mathbb{N}}$ strictement décroissante pour $\preceq$. $u_{n+1} \preceq u_n$ et $u_{n+1} \neq u_n$ donc $|u_{n+1}| < |u_n|$. $(|u_n|)_{n\in\mathbb{N}}$ est une suite infinie strictement décroissante pour $\leq$ dans $\mathbb{N}$ qui est bien fondé. C'est absurde donc aucune telle suite $(u_n)_{n\in\mathbb{N}}$ n'existe, et $(\Sigma^*, \preceq)$ est un ensemble bien fondé.
$\preceq$ n'est pas total car ab et ba ne sont pas des mots comparables.

On rappelle que $u^0 = \varepsilon$ est le mot vide et que $u^{k+1} = uu^k$, c'est-à-dire que $u^k = uu\dots u$, avec $|u^k| = k|u|$.

☐ 3 – Montrer que pour tout mot p, il existe un mot u tel que p est préfixe de u^k pour un certain $k \in \mathbb{N}$ et justifier qu'il existe un tel mot u de longueur minimale. On appelle *période* la longueur d'un tel u.

Réponse 3. Soit X l'ensemble des mots u tels que p est préfixe de u^k pour un certain $k \in \mathbb{N}$. Cet ensemble est non-vide car $p \in X$: p est préfixe de p^1.
Comme $\preceq$ est bien fondé, X admet un élément minimal pour $\preceq$, c'est-à-dire un tel u de longueur minimale.

☐ 4 – Donner en justifiant les périodes de *aaab*, de *abcabcab* et de *ababa*.

Réponse 4. La période de *aaab* est 4, un mot plus court ne pourrait contenir de b.
La période de *abcabcab* est 3 car *abcabcab* est préfixe de $(abc)^3$ et un mot plus court ne contiendrait pas de c.
La période de *ababa* est 2 car *ababa* est préfixe de $(ab)^3$ et un mot plus court ne contiendrait pas de b.

☐ 5 – Après une occurence à l'indice i de chacun de ces motifs, à partir de quel indice doit-on chercher la prochaine occurence ?

Réponse 5. La prochaine occurence doit se chercher à l'indice $i + d$, avec d la période du motif cherché. Donc pour *aaab* en $i + 4$, pour *abcabcab* en $i + 3$ et pour *ababa* en $i + 2$.

☐ 6 – Si on vient de trouver le motif a^m de période 1 en position i. Combien de lettres doit-on vérifier pour savoir si il apparaît aussi en position $i + 1$? généraliser le raisonnement à tout mot u.

Réponse 6. Il n'y a qu'une seule nouvelle lettre à tester. En effet, on sait que les $m - 1$ premières lettres sont correctes. Dans le cas général, il faut tester d lettres où d est la période du mot.

Une autre manière de définir le décalage que l'on cherche à produire est la suivante : on cherche la taille du plus grand suffixe propre de p qui est en est aussi un préfixe. Un suffixe propre de p étant un suffixe de p différent de p.

☐ 7 – Réécrire la fonction **int** strlen(**char** w[]) qui calcule la taille d'une chaîne de caractères.

Réponse 7.

C

```
int strlen(char w[]){
  int cpt = 0;
  while (w[cpt] != '\0') cpt = cpt + 1;
  return cpt;
}
```

☐ 8 – Écrire une fonction bool is_suffix_prefix(**char** w[], **int** i) qui prend en entrée un mot w et un entier $i < |w|$ (ce que l'on ne vérifiera pas), et qui renvoie true si le suffixe de taille i de w et son préfixe de taille i sont égaux, et false sinon.

Réponse 8.

C

```
bool is_suffix_prefix(char w[], int i){
  int n = strlen(w);
  for (int j = 0; j < i; j = j + 1)
    if (w[j] != w[n - i + j]) return false;
  return true;
}
```

☐ 9 – Déduire des deux questions précédentes une fonction **int** galil (**char** w[]) qui renvoie le plus grand entier k tel que le suffixe de taille k de w et son préfixe de taille k sont égaux.

Réponse 9.

C

```
int galil(char w[]){
  for (int k = strlen(w) - 1; k > 0; k = k + 1)
    if (is_suffix_prefix(w, k)) return k;
  return 0;
}
```

☐ 10 – Donner en justifiant brièvement la complexité temporelle dans le pire des cas de la fonction galil en utilisant une notation de Landau $O(\cdot)$.

Réponse 10. La complexité de strlen est $O(|w|)$, celle de is_suffix_prefix est donc $O(|w|)$ et celle de *galil* est donc $O(|w|^2)$, la somme des entiers de 0 à $|w|$.

Cette valeur peut aussi être calculée en $O(|w|)$. Pour ce faire, nous allons calculer un tableau π donnant pour chaque indice i allant de 0 à $|w|-1$, la taille du plus grand suffixe propre de $w_0 \dots w_i$ qui est aussi un préfixe de $w_0 \dots w_i$, où $w_0 \dots w_i$ est le préfixe de w de taille $i+1$.

☐ 11 – Quelle est la valeur de $\pi[0]$?

Réponse 11. Le seul suffixe propre de w_0 est ε, donc $\pi[0] = 0$.

☐ 12 – Montrer que pour tout i allant de 0 à $|w|-2$, $\pi[i+1] \leq \pi[i]+1$, et donner la condition pour que $\pi[i+1] = \pi[i]+1$.

Réponse 12. On a $w_0 \dots w_{\pi[i+1]-1} = w_{i+2-\pi[i+1]} \dots w_{i+1}$.
Donc $w_0 \dots w_{\pi[i+1]-2} = w_{i+2-\pi[i+1]} \dots w_i$, c'est-à-dire que le suffixe propre de $w_0 \dots w_i$ de taille $\pi[i+1]-1$ est aussi un préfixe de $w_0 \dots w_i$.
Donc $\pi[i] \geq \pi[i+1]-1$ par définition de $\pi[i]$.
On a égalité si et seulement si $w_{\pi[i]} = w_{i+1}$, car dans ce cas, on a $w_0 \dots w_{\pi[i]} = w_{i+1-\pi[i]} \dots w_{i+1}$, et $\pi[i+1] = \pi[i]+1$ par double inégalité.

Pour un i fixé, posons $j = \pi[i-1]$. On a par définition $w_{i-j} \dots w_{i-1} = w_0 \dots w_{j-1}$. Lorsque $w_i \neq w_j$, on a donc le plus grand suffixe propre et préfixe de $w_0 \dots w_i$ qui est également le plus grand suffixe propre et préfixe de $w_0 \dots w_{j-1} w_i$. On peut alors tester l'égalité entre w_i et w_j, et s'il ne sont pas égaux, continuer avec $\pi[j-1]$, etc.

☐ 13 – Écrire une fonction **int** ∗prefixe(**char** w[]) qui calcule et renvoie le tableau π correspondant au mot p en utilisant la stratégie précédente.

Réponse 13.

C

```c
int *prefixe(char w[]){
  int n = strlen(w);
  int *pi = malloc(n * sizeof(int));
  pi[0] = 0;
  for (int i = 1; i < n; i = i + 1){
    int j = pi[i-1];
    while (w[j] != w[i] && j > 0){
      j = pi[j-1];
    }
    if (w[j] == w[i]) pi[i] = j + 1;
    else              pi[i] = 0;
  }
  return pi;
}
```

☐ 14 – Montrer que la complexité de cette fonction est $O(|w|)$ dans le pire des cas.
On pourra démontrer que si on doit effectuer k tests différents pour calculer $\pi[i]$, alors on peut les répartir sur des indices précédents qui n'ont effectués qu'un seul test pour montrer que la moyenne des tests par indice est bornée par une constante.

Réponse 14. https://www.youtube.com/watch?v=nJbNe0Yzjhw

Après avoir calculé cette table, on sait qu'après avoir trouvé une occurence dans l'algorithme de Boyer-Moore, il nous suffit de décaler le motif de $\pi[|p|-1]$ emplacements, et de vérifier seulement si les $|p| - \pi[|p|]$ derniers caractères sont égaux à ceux du texte.

☐ 15 – Implémenter en C l'algorithme de Boyer-Moore avec la règle de Galil pour gérer les occurences trouvées.
Pour cette question, toute trace de recherche, même incomplète, sera prise en compte lors de l'évaluation.

Réponse 15. On met bout à bout les éléments vus précédemment.
La différence majeure à noter est que la boucle d ecomparaison (sur j) ne va plus nécessairement jusque 0, mais peut s'arrêter avant si on avait trouvé une occurence.
(réponse non testée, générée par un LLM (large language model).

C

```c
int search_galil(char pattern[], char text[]){
  int cpt = 0;
  int n = strlen(text);
  int m = strlen(pattern);

  int **table = bad_char(pattern);
  int *pi = prefixe(pattern);
  int shift = 0;

  for (int i = 0; i <= n - m;){
    int j;
    for (j = m - 1; j >= shift; j--){
      unsigned char c = text[i + j];
      if (c != pattern[j]){
        i += table[j][c];
        shift = 0;
        break;
      }
    }
    if (j < shift){
      cpt++;
      printf("Occurence en position %d\n", i);
```

```
            shift = pi[m - 1];
            i += (m - shift);
        }
    }

    free_bad_char(table, m);
    free(pi);
    return cpt;
}
```

On peut montrer qu'avec cette règle, l'algorithme de Boyer-Moore termine sa recherche en $O(|t|)$ dans le pire des cas si on suppose que l'étape de précalcul est de complexité négligeable devant $O(|t|)$.

2 Recherche dans une matrice triée – C – (35 pts)

Dans cette partie, on s'intéresse à la recherche d'un élément x dans une matrice $A = (a_{i,j})$ de dimension $n \times m$ dont chaque ligne et chaque colonne sont triés dans l'ordre croissant, i.e. $\forall i_1, i_2, j_1, j_2$ tels que $i_1 \leq i_2, j_1 \leq j_2$, on a $a_{i_1,j_1} \leq a_{i_2,j_1}$ et $a_{i_1,j_1} \leq a_{i_1,j_2}$.
On souhaite implémenter la solution en C. La matrice est représentée par un tableau de tableaux. En voici un exemple :

```
int mat[4][4] =
    {{-12,  5,  8,  7},
     { -4, 10, 20, 27},
     {  0, 24, 32, 54},
     { 70, 72, 73, 96}
    };
```

Dans un premier temps, nous simplifions le problème en considérant une matrice A carrée, de dimension $n \times n$, avec $n = 2^p$, $p \in \mathbb{N}$.

☐ 16 – Écrire une fonction bool check(**int** n, **int** **mat) qui prend en entrée une matrice et vérifie que chacune de ses lignes et chacun de ses colonnes est triée dans l'ordre croissant..

Réponse 16. On vérifie séparément les lignes puis les colonnes, par exemple :

C

```
bool check(int n, int **mat) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - 1; j++) {
            if (mat[i][j] > mat[i][j + 1]) return false;
        }
    }
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n - 1; i++) {
            if (mat[i][j] > mat[i + 1][j]) return false;
        }
    }
    return true;
}
```

☐ 17 – Montrer que les éléments $a_{i,i}$ sur la diagonale descendente sont triés dans l'ordre croissant.

Réponse 17. Puisque chaque ligne et chaque colonne sont triées dans l'ordre croissant, nous avons : - $a_{i,i} \leq a_{i+1,i}$ car la colonne i est triée. - $a_{i+1,i} \leq a_{i+1,i+1}$ car la ligne $i+1$ est triée. Par transitivité, on obtient $a_{i,i} \leq a_{i+1,i+1}$, ce qui prouve que les éléments diagonaux sont triés.

☐ 18 – Écrire une fonction **int** find_diag(**int** n, **int** ∗∗mat) qui prend en entrée un entier n et une matrice A de dimension $n \times n$ et qui renvoie l'indice k tel que $a_{k,k} \leq x \leq a_{k+1,k+1}$ à partir de x. La complexité dans le pire des cas de la fonction doit être $O(\log n)$.

Réponse 18. Une recherche dichotomique permet d'obtenir cet indice en $O(\log n)$:

C

```c
int find_diag(int n, int **mat, int x) {
    int left = -1, right = n;

    while (left + 1 < right) {
      // invariant : pour tout i <= left , mat[i][i] <= x
      //             pour tout j >= right , mat[j][j] > x
        int mid = (left + right) / 2;

        if (mat[mid][mid] <= x) {
            left = mid;
        } else {
            right = mid;
        }
    }

    return left;
}
```

☐ 19 – Est-il possible qu'un tel entier k n'existe pas ? Que peut-on affirmer le cas échéant ?

Réponse 19. Oui, si x est strictement inférieur à $a_{0,0}$ ou strictement supérieur à $a_{n-1,n-1}$, alors il n'existe pas d'indice k tel que $a_{k,k} \leq x \leq a_{k+1,k+1}$.
Dans ce cas, on peut conclure que x n'est pas présent dans la matrice.

☐ 20 – Si $x = a_{k,k}$ ou $x = a_{k+1,k+1}$, alors la recherche se termine, on considère maintenant qu'on a $a_{k,k} < x < a_{k+1,k+1}$. Dessiner la matrice A et indiquer quels sont les coefficients dont on sait qu'ils sont strictement inférieurs à x et ceux dont on sait qu'ils sont strictement supérieurs à x.

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,k} & a_{1,k+1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{k,1} & \cdots & a_{k,k} & a_{k,k+1} & \cdots & a_{k,m} \\ a_{k+1,1} & \cdots & a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,k} & a_{n,k+1} & \cdots & a_{n,m} \end{pmatrix}$$

Réponse 20. À gauche et en haut de $a_{k,k}$, les éléments sont strictement plus petits que x. À droite et en bas de $a_{k+1,k+1}$, les éléments sont strictement plus grands que x.

Nous souhaitons à présent étudier les cas extrêmes.

☐ 21 – Combien de cases reste-t-il à explorer si $k = 0$? Quelle est alors la complexité de la recherche de x dans ces cases, et par quelle méthode ? Détailler.
Même question pour $k = n - 1$.

Réponse 21. Si $k = 0$, il reste une ligne et une colonne de $n - 1$ cases à vérifier, chacune peut se faire par une recherche dichotomique en $O(\log n)$.
Le cas $k = n - 1$ est similaire et se traite par la même méthode avec la même complexité.

☐ 22 – Déterminer en justifiant la valeur de k pour laquelle le nombre de cases dans lesquelles peut se trouver x est maximale, c'est-à-dire, la valeur de k pour laquelle le nombre de coefficients indiqués en question 20 est minimal.

Réponse 22. Dans le cas général, il reste $2(n-k)k$ cases restantes. Il s'agit d'une fonction polynôme du second degré en k qui est maximale lorsque sa dérivée $2n-4k$ est nulle, c'est-à-dire lorsque $k=\frac{n}{2}$. Si cette valeur n'est pas entière, ses parties entières supérieures et inférieures atteignent toutes deux le maximum.

On se retrouve pour cette valeur de k à faire une recherche dans deux matrices de dimension $2^{p-1} \times 2^{p-1}$. Si on suppose que ce cas est le pire des cas, on a besoin pour déterminer la complexité dans le pire des cas de notre algorithme de résoudre la relation de récurrence suivante :

$$T(0) = 1$$
$$T(p) = 2T(p-1) + p$$

☐ 23 – On pose $T'(p) = T(p) + p$. Donner une relation de récurrence vérifiée par T'.

Réponse 23. $T'(p) = T(p) + p = 2T(p-1) + 2p = 2T'(p-1) + 2$.

☐ 24 – Montrer que $T(p) = O(2^p)$.

Réponse 24. Si on pose $T''(p) = T'(p) + 2$, on a $T''(p) = T'(p) + 2 = 2T'(p-1) + 4 = 2T''(p-1)$, et $T''(p) = 2^p T''(0)$ car c'est une suite géométrique de raison 2, avec $T''(0) = 3$.
On a alors $T'(p) = 3 \times 2^p + 2$ et $T(p) = 3 \times 2^p + p + 2 = O(2^p)$.

☐ 25 – En déduire la complexité dans le pire des cas de cette approche en fonction de n.

Réponse 25. $p = \log(n)$ donc $2^p = n$, la complexité de cette approche est $O(n)$.

☐ 26 – Proposer une généralisation de la méthode aux matrices qui ne sont pas carrés (on pourra s'intéresser à la diagonale qui passe par le coefficient central de la matrice) et l'implémenter en C. Évaluer sa complexité.
Pour cette question, toute trace de recherche, même incomplète, sera prise en compte dans l'évaluation.

Réponse 26. Un algorithme récursif est le plus simple à coder car chaque problème nécessite la résolution de 2 sous-problèmes. On peut prendre 4 arguments en plus de la matrice pour la zone délimitée, de manière similaire à la recherche dichotomique.
Les cas de base sont des lignes ou des colonnes, il faut alors invoquer une recherche dichotomique.

3 Couple de points les plus proches – OCaml – (25 pts)

Dans cette partie, nous allons nous intéresser au problème suivant :

Parmi une liste de points de $\mathbb{N}^2$ (représentés par le type int $*$ int),
on cherche à trouver le couple de points les plus proches (en distance euclidienne).

Nous allons d'abord étudier ce problème avant de proposer une solution par recherche exhaustive à l'aide d'une droite de balayage. Soit $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ la distance euclidienne.

☐ 27 – Combien de couples de points différents peut-on former avec n points ?

Réponse 27. $\binom{n}{2} = \frac{n(n-1)}{2}$.

☐ 28 – Soit $\preceq$ la relation binaire sur $\left(\mathbb{N}^2\right)^2$ suivante :

$$(p, q) \preceq (p', q') \text{ si et seulement si } d(p, q) \leq d(p', q').$$

$\preceq$ est-elle une relation d'ordre ? totale ? bien-fondée ?

Réponse 28. $(p, q) \preceq (q, p)$ et $(q, p) \preceq (p, q)$ donc cette relation n'est pas antisymétrique. Ce n'est pas une relation d'ordre.

☐ 29 – Écrire une fonction sqr_dist qui prend en entrée un couples de points (p, q) et qui calcule $d(p, q)^2$.
Le type de la fonction n'est pas imposée. Elle peut prendre en entrée 2 points, ou alors un couple de points, chaque point étant un couple d'entiers.

Réponse 29.

```
let dist ((x1,y1),(x2,y2)) =
  let dx, dy = x2-x1, y2-y1 in
  dx*dx+dy*dy
```

☐ 30 – Écrire une fonction plus_proches qui prend en entrée 2 couples de points (p, q) et (p', q') et qui teste si $(p, q) \preceq (p', q')$.
Le type de la fonction n'est pas imposée. Elle peut prendre en entrée 4 points, ou alors deux couples de points, chaque point étant un couple d'entiers.

Réponse 30.

```
let plus_proches (p1,q1) (p2,q2) =
  dist (p1, q1) < dist (p2, q2)
```

Nous allons à présent parcourir la liste des points avec une droite de balayage, c'est-à-dire que nous considérons que les points sont triés dans l'ordre lexicographique.

☐ 31 – Rappeler la définition de l'ordre lexicographique. Est-il bien fondé ? total ? (on ne demande pas de démonstration pour cette question)

Réponse 31.

$$(x, y) \leq (x', y') \text{ si et seulement si } x > x' \text{ ou } (x = x' \text{ et } y \leq y')$$

C'est un ordre bien fondé et total.

Notre recherche exhaustive se passe de la manière suivante : on parcourt l'ensemble des points, en mémorisant le couple (p, q) de points le plus proches que l'on a rencontré. À chaque nouveau point (x, y), nous considérons les couples formés de ce point et de tous les points précédents.

☐ 32 – Que peut-on dire de l'abscisse des points qui se trouvent à une distance à (x, y) inférieure à $d(p, q)$?

Réponse 32. Si le point à tester (x', y') possède une abscisse inférieure ou égale à $x - d(p, q)$, alors on sait que sa distance au point (x', y') est nécessairement trop grande. On peut donc ne regarder que la partie de la liste qui possède une abscisse supérieure strictement à $x - d(p, q)$.

☐ 33 – Écrire une fonction récursive superieurs_a : int -> (int * int) list -> int list qui prend en entrée un entier k et une liste ℓ de points triés dans l'ordre croissant, et qui renvoit la sous-liste ℓ' de ℓ qui ne contient que les éléments dont l'abscisse est supérieure (ou supérieure ou égale) à k (les éléments dont l'abscisse est égale à k peuvent être soit tous inclus, soit tous exclus, soit certains inclus et d'autres exclus).

Réponse 33.

```
let rec superieurs_a k l = match l with
  | [] -> []
  | t::q -> if t > (k,0) then t::q
            else superieurs_a k q;;
```

Cette recherche n'est pas très efficace, car il faut quand même parcourir tous les éléments de la liste pour les défausser, et on ne profite pas au maximum du fait que la liste soit triée.
Pour améliorer la complexité de notre algorithme, nous choisissons plutôt de mettre nos points dans un tableau afin d'avoir un accès en temps constant à chacun des éléments.

☐ 34 – Écrire une fonction superieurs_a_dic : int -> (int * int) array -> int qui prend en entrée un entier k et un tableau trié t, et qui renvoie le plus petit indice i tel que $t.(i)$ possède une abscisse supérieure (ou supérieure ou égale) à k en procédant par dichotomie.
La complexité de l'algorithme doit être logarithmique en la taille du tableau, on ne demande pas de le démontrer.

Réponse 34.

```
let superieurs_a_dic k t =
  let n = Array.length t in
  let rec cherche a b =
    if b - a <= 1 then b
    else let c = (a+b)/2 in
         if t.(c) < (k,0) then
           cherche c b
         else cherche a c
  in cherche 0 (n-1);;
```

☐ 35 – Il est intéressant de noter que lors de notre parcours, on peut stocker dans une liste les points déjà passés, en ajoutant à chaque étape le point que l'on vient de tester. Cette liste sera-t-elle triée dans l'ordre croissant ou dans l'ordre décroissant ? Est-ce plus intéressant ? Justifier.

Réponse 35. Le dernier point ajouté à la liste est plus grand que tous les précédents, donc la liste est triée dans l'ordre décroissant si on parcourt les points dans l'ordre croissant. Cet ordre est plus intéressant car on peut s'arrêter dès que l'abscisse devient trop petite, et on a alors parcouru les couples qui nous intéressent.

☐ 36 – Après avoir choisi entre une représentation de notre ensemble de points par une liste ou par un tableau, écrire une fonction plus_proche_paire qui prend en entrée une liste de points et renvoie la paire de points dont la distance est minimale.
Pour cette question, toute trace de recherche, même incomplète, sera prise en compte dans la notation. Tout code doit être expliqué.

Réponse 36.

style fonctionnel

```
let plus_proche_paire l =
  match l with
  | [] | [_] -> failwith "Pas assez de points"
  | t1::t2::q ->
     let rec parcours best previous = function
       | [] -> best
       | (x,y)::q ->
          let k = d best in
          parcours
            (parcours_filtre (x,y) (x-k) best previous)
            ((x,y)::previous) q
     and parcours_filtre p k best = function
       | [] -> best
       | (x,y)::q ->
          if x < k then best
          else if plus_proches (p,(x,y)) best then
            parcours_filtre p k (p,(x,y)) q
          else parcours_filtre p k best q
     in parcours (t1,t2) [t2;t1] q;;
```

style impératif

```
let plus_proche_paire tab =
  let n = Array.length tab in
  assert (n >= 2);
  let best = ref (tab.(0),tab.(1)) in
  for i = 2 to n-1 do
    let (x,y) = tab.(i) in
    let k = d !best in
    for j = (superieurs_a_dic (x-k) tab) to i-1 do
      if plus_proches (tab.(i),tab.(j)) !best then
        best := (tab.(i),tab.(j))
    done
  done; !best;;
```

4 Problème des huit dames – (15 pts)

Le but du problème des huit dames est de placer huit dames d'un jeu d'échecs sur un échiquier de 8×8 cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée). Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale.
Ce problème est à traiter au choix en C, en OCaml, ou en pseudocode. L'utilisation d'un des langages de programmation, C ou OCaml sera valorisée.
On représente un échiquier par un tableau de 64 cases, numérotées de 0 à 63, la case 0 étant le coin en haut à gauche et la case 63 le coin en bas à droite. Les colonnes et les lignes sont numérotées de 0 à 7.

☐ 37 –

1. Comment trouver le numéro de colonne c d'une case dont on connaît le numéro n ?
2. Comment trouver le numéro de ligne r d'une case dont on connaît le numéro n ?

Les réponses à ces questions donnent deux fonctions `col` et `row` que l'on utilisera plus tard.

Réponse 37.

C

```c
int col(int n){
        return n % 8;
}
int row(int n){
        return n / 8;
}
```

```
let col n = n mod 8;;
let row n = n / 8;;
```

☐ 38 – Proposer une numérotation des diagonales. Comment trouver le numéro de la diagonale sur laquelle se trouve une case ?
On pourra traiter uniquement les diagonales montantes ou les diagonales descendantes, l'autre cas étant similaire.

Réponse 38. On peut numéroté les diagonales montantes de 0 à 14 en ajoutant le numéro de ligne et le numéro de colonne par exemple :

C

```c
int diag_m(int n){
        return col(n) + row(n);
}
```

```
let diag_m n = col n + row n;;
```

☐ 39 – Un élève propose l'algorithme suivant en pseudocode pour vérifier une solution au problème des huit dames. On suppose que la solution est donnée sous forme d'une liste/d'un tableau qui indique les cases dans lesquelles sont placés des dames :

```
Input : sol
Pour i = 0 .. taille(sol) :
  Pour j = i+1 .. taille(sol) :
    si col(sol[i]) == col(sol[j]) :
```

```
      return false
    si row(sol[i]) == row(sol[j]) :
      return false
return true
```

Cette fonction est fausse car incomplète. Elle retourne `true` pour certaines solutions qui ne sont en fait pas correctes. Proposer trois tests qui montrent trois défauts différents de cette fonction. Préciser les données en entrée et le résultat attendu, et commenter en quoi vos trois tests vérifient des propriétés différentes.

Réponse 39.

1. $[]$, résultat attendu faux. L'algorithme proposé ne vérifie pas que la solution comporte 8 dames.
2. $[-1; -8; 72; 89; 45; -4; -6; 0]$, résultat attendu faux. L'algorithme proposé ne vérifie pas que les numéro de cases de la solution sont bien entre 0 et 63.
3. $[0; 9; 18; 27; 36; 45; 54; 63]$, résultat attendu faux. L'algorithme proposé ne vérifie pas que deux dames ne sont pas sur la même diagonale.

☐ 40 – On suppose que la fonction précédente a été corrigée, et est disponible sous le nom de `check_sol`, dont vous pourrez choisir le type en fonction du langage de programmation que vous avez choisi.
Donner le nom d'une famille d'algorithmes qui permet d'améliorer la recherche exhaustive dans le cas d'un problème de contraintes, puis écrire un algorithme qui permet de résoudre le problème des huit dames.

Réponse 40. On peut résoudre aisément ce problème à l'aide d'un algorithme de retour sur trace (backtracking).

```
exception Found of int list;;

let solve () =
  let rec parcours acc i =
    if i = 63 && check_sol acc then
      raise (Found acc)
    else if i = 63 then ()
    else
      parcours acc (i+1); parcours (i::acc) (i+1)
  in try parcours [] 0; [] with Found sol -> sol;;
```

Fin de l'épreuve