A - Préliminaires

Question 1.

```c
int plus_bas(point *tab, int n)
{
  int bas = 0;
  for (int i = 1; i < n; i++)
  {
    if (tab[i].y < tab[bas].y || (tab[i].y == tab[bas].y && tab[i].x < tab[bas].x))
    {
      bas = i;
    }
  }
  return bas;
}
```

Question 2. On veut en fait trouver le minimum d'un tableau pour ce qu'on appelle l'ordre lexicographique. On peut montrer que c'est une relation d'ordre.

Inv(i) : `bas` est l'indice du plus petit point pour cet ordre parmi les i premiers points de `tab`.

Preuve :

- Inv(1) est vrai avant la boucle car bas est l'indice du premier point ;
- Si Inv(i) est vrai en haut de la boucle, alors on teste si `tab[i] < tab[bas]` pour l'ordre lexicographique :
  - si le est est vérifié, alors `tab[i] < tab[bas] < tab[j]` pour $j \in [\![0, i-1]\!]$ (par HR), donc (par transitivité de la relation d'ordre), l'élément d'indice i est bien plus petit que les $i+1$ premiers éléments de `tab` ;
  - sinon, alors `tab[bas]` est bien plus petit que les $i+1$ premiers éléments de `tab` (par HR).

  Donc Inv($i+1$) est vrai en bas de la boucle.

Donc Inv(i) est bien un invariant de boucle, et après la boucle $i = n$), Inv(n) est vrai : la fonction est correcte.

Question 3.

1. $i = 0$, $j = 3$, $k = 4$: $orient = +1$.
2. $i = 8$, $j = 9$, $k = 10$: $orient = -1$.

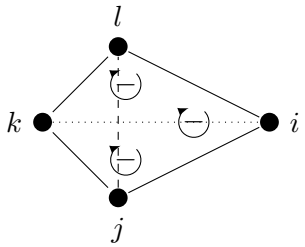
Question 4.

```c
int orient(point *tab, int i, int j, int k)
{
  int aire = (tab[k].y - tab[i].y) * (tab[j].x - tab[i].x)
    - (tab[k].x - tab[i].x) * (tab[j].y - tab[i].y);
  if (aire > 0) { return 1; }
  else if (aire < 0) { return -1; }
  else { return 0; }
}
```

B - Algorithme du paquet cadeau

Question 5.

- refléxive : on a bien $p_j \preceq p_j$ car `orient(tab, i, j, j)` $= 0$;
- antisymétrie : si $p_j \preceq p_k$ et $p_k \preceq p_j$, alors `orient(tab, i, j, k)` $= 0$, donc $p_j = p_k$;
- antisymétrie : sur un dessin (on demande de justifier brièvement), on se rend compte que si (p_i, p_j, k) et (p_i, p_k, p_l) sont orientés négativement, alors il en est de même pour (p_i, p_j, p_l). Donc $p_j \preceq p_k \preceq p_l \Rightarrow p_j \preceq p_l$.



Question 6.

```
int prochain_point(point *tab, int n, int i)
{
  int j = 0;
  if (i == 0) // il ne faut jamais avoir i == j
  {
    j = 1;
  }
  for (int k = 0; k < n; k++)
  {
    if (k != i && k != j) // trois points distincts
    {
      if (orient(tab, i, j, k) <= 0)
      {
        j = k;
      }
    }
  }
  return j;
}
```

Question 7. Au début, $j = 0$ et k parcourt tous les entiers dans $[\![0, 11]\!]$.

- pour $k = 1$, `orient(10,0,1)` ≤ 0 donc $j = 1$;
- pour $k = 2$, `orient(10,1,2)` ≤ 0 donc $j = 2$;
- pour $k = 5$, `orient(10,2,5)` ≤ 0 donc $j = 5$.

Ensuite, la valeur de j n'est plus modifiée.

Question 8.

```c
int conv_jarvis(point *tab, int n, int *env)
{
  int i = plus_bas(tab, n); // O(n)
  int j = prochain_point(tab, n, i); // O(n)
  int nb = 0;
  env[nb] = i;
  nb++;
  while (j != i) // On passe au plus n fois dans cette boucle
  {              // donc complexité en O(n^2)
    env[nb] = j;
    nb++;
    j = prochain_point(tab, n, j); // O(n)
  }
  return nb;
}
```

Question 9. On passe autant de fois dans la boucle `while` qu'il y a de points dans l'enveloppe convexe, soit au plus n fois. La complexité est donc en $O(n^2)$.

C - Intermède : piles d'entiers

D - Algorithme de balayage

Question 10.

```c
void maj_es(point *tab, stack es, int i)
{
  int s1 = pop(es);
  if (!is_empty(es))
  {
    int s2 = top(es);
    while (!is_empty(es) && orient(tab, i, s1, s2) < 0)
    {
      s1 = pop(es);
      if (!is_empty(es)) { s2 = top(es); }
    }
  }
  push(s1, es);
  push(i, es);
}
```

Question 11.

```c
void maj_ei(point *tab, stack ei, int i)
{
  int s1 = pop(ei);
  if (!is_empty(ei))
  {
    int s2 = top(ei);
    while (!is_empty(ei) && orient(tab, i, s1, s2) > 0) // et pas < 0 cette fois !
    {
      s1 = pop(ei);
      if (!is_empty(ei)) { s2 = top(ei); }
    }
  }
  push(s1, ei);
  push(i, ei);
}
```

Question 12.

```c
stack conv_graham(point *tab, int n)
{
  stack es = new_stack();
  stack ei = new_stack();
  push(0, es);
  push(0, ei);
  for (int i = 1; i < n; i++)
  {
    maj_es(tab, es, i);
    maj_ei(tab, ei, i);
  }
  /* fusion de es et ei :
     on vide es dans ei, sauf son premier et dernier élément */
  pop(es); // on enlève le dernier élément
  while (!is_empty(es)) // on vide es dans ei
  {
    push(pop(es), ei);
  }
  pop(ei); // on enlève le premier élément de es
  return ei;
}
```

Question 13.

- Chaque opération sur les piles ou appel à `orient` se fait en temps constant.
- Les fonctions `maj_es` et `maj_ei` ont une complexité variable selon la situation, mais prenons un peu de recul :
  - lors de l'exécution de la boucle `for`, chaque point du nuage entre une seule fois dans chacune des piles `ei` et `es`, et n'en sort que zéro ou une fois ;
  - cette boucle va donc s'exécuter en $O(n)$.

  - la boucle `while` qui vide `es` dans `ei` s'exécute également en $O(n)$.

Donc la complexité de la fonction `conv_graham` est en $O(n)$.