Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`, `<stdassert.h>`, ...) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Questions de cours : terminaison*

On considère la fonction d'Ackerman $a : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ définie par par :

$$\begin{cases} a(0,m) &= m+1 \\ a(n,0) &= a(n-1,1) \text{ si } n > 0 \\ a(n,m) &= a(n-1,a(n,m-1)) \text{ si } n > 0 \text{ et } m > 0 \end{cases}$$

Q1– Calculer $a(1,2)$

$$\begin{aligned} a(1,2) &= a(0,a(1,1)) \\ &= a(0,a(0,a(1,0))) \\ &= a(0,a(0,a(0,1))) \\ &= a(0,a(0,2)) \\ &= a(0,3) \\ &= 4 \end{aligned}$$

Q2– Ecrire en OCaml une fonction `ack: int -> int -> int` qui prend en argument deux entiers positifs n et m et renvoie $a(n,m)$.

```
let rec ack n m =
  match n, m with
  | 0, m -> m + 1
  | n, 0 -> ack (n - 1) 1
  | n, m -> ack (n - 1) (ack n (m - 1));;
```

Q3– Prouver la terminaison de la fonction `ack` on précisera soigneusement le variant et la relation d'ordre bien fondée utilisée.

On considère l'ordre lexicographique sur N^2 noté $\preccurlyeq_L$, montrons que (n,m) est un variant .

- si $m = 0$ alors on effectue un appel récursif avec $(n-1,1)$ et comme $(n-1,1) \preccurlyeq_L (n,0)$ la quantité (n,m) décroit strictement.
- sinon, on effectue un premier appel récursif avec $(n,m-1)$ et comme $(n,m-1) \preccurlyeq_L (n,m)$ la quantité (n,m) décroit strictement. Le second appel récursif s'effectue avec $(n-1,a(n,m-1))$ qui est strictement inférieur à (n,m).

Dans tous les cas (n,m) décroit strictement à chaque appel récursif, c'est donc un variant et puisque $(N^2,\preccurlyeq)$ est un ensemble bien fondé, la fonction `ack` termine.

❑ Exercice 2 : *Questions de cours : preuve par induction structurelle*

Q4– Donner la définition inductive des arbres binaires sur un ensemble d'étiquettes E.

On définit inductivement l'ensemble des arbres binaires sur un ensemble d'étiquettes E avec :
— L'ensemble d'axiomes $X_0 = \{\varnothing\}$ où $\varnothing$ est l'arbre vide.
— La règle d'inférence d'arité 2 : $r : (g, d) \to (g, e, d)$ où $e \in E$.

Q5– Ecrire en OCaml un type `arbrebin` représentant un arbre binaire en utilisant un type paramétré `'a` pour l'ensemble des étiquettes.

```
type 'a arbrebin =
  | Vide
  | Noeud of  'a arbrebin * 'a * 'a arbrebin
```

Q6– Rappeler la définition de la hauteur et de la taille d'un arbre binaire.

- Le nombre de noeuds d'un arbre binaire A, noté $n(A)$, se définit récursivement par :
$$\begin{cases} n(A) = 0 & \text{si } A \text{ est vide} \\ n(A) = 1 + n(g) + n(d) & \text{si } A = r(g, d) \end{cases}$$
- La hauteur d'un arbre binaire A, noté $h(A)$, se définit récursivement par :
$$\begin{cases} h(A) = -1 & \text{si } A \text{ est vide} \\ h(A) = 1 + \max(h(g), h(d)) & \text{si } A = r(g, d) \end{cases}$$

Q7– Prouver *par induction structurelle* que la taille d'un arbre binaire de hauteur h est inférieur ou égale à $2^{h+1} - 1$.

Soit A un arbre binaire de hauteur n et de taille h on note $\mathcal{P}(A)$ la propriété $n \leqslant 2^{h+1} - 1$. Montrons par induction structurelle que P est vraie pour tous les arbres binaires.
— $\mathcal{P}$ est vraie pour tout axiome, en effet il y a un seul axiome, l'arbre vide qui par définition est de taille 0 et de hauteur -1 et on a bien $2^{-1+1} - 1 \geqslant 0$.
— Supposons que $\mathcal{P}$ est vraie pour deux arbres g et d et montrons qu'alors $\mathcal{P}(r(g, d))$ est vraie, c'est à dire la conservation de $\mathcal{P}$ par application de l'unique règle d'inférence.

$$\begin{array}{rcll} n & = & n(g) + n(d) + 1 & \text{par définition de la taille de } r(g,d) \\ n & \leqslant & 2^{h(g)+1} - 1 + 2^{h(g)+1} - 1 + 1 & \text{car } P(g) \text{ et } P(d) \text{ sont vraies} \\ n & \leqslant & 2^{\max(h(g),h(d))+1} - 1 + 2^{\max(h(g),h(d))+1} - 1 + 1 & \\ n & \leqslant & 2^{\max(h(g),h(d))+2} - 1 & \\ n & \leqslant & 2^{h+1} - 1 & \end{array}$$

Donc par induction structurelle, $\mathcal{P}$ est vraie pour tout arbre binaire.

❑ Exercice 3 : *Recherche des k premiers maximums d'une liste*

Les fonctions demandées dans cet exercice doivent être écrites en langage OCaml.

On s'intéresse au problème de la recherche des k premiers maximums d'une liste de n entiers. Dans toute la suite de l'exercice on supposera que la liste est *non vide* : $n > 0$ et qu'on extrait moins de maximums qu'il n'y a d'éléments dans la liste c'est à dire que $k \leqslant n$. On cherche donc à écrire une fonction `kmax : int list -> int -> int list` qui renvoie la liste des `k` premiers maximums de la liste donnée en argument. Par exemples :

— `kmax [2; 5; 1; 8; 3; 0; 4] 2` renvoie `[8, 5]`
— `kmax [7; 8; 8; 1; 6; 3; 2; 9] 3` renvoie `[9; 8; 8]`
— `kmax [1; 0; 1; 2; 4] 4` renvoie `[4; 2; 1; 1]`

Les trois parties sont indépendantes et dans chacune d'elle on propose un algorithme différent afin d'écrire la fonction `kmax`.

■ Partie I : Résolution par recherche successive des maximums

Q8– Ecrire une fonction `max_reste : int list -> int * int list` qui prend en argument une liste et renvoie le couple composé du maximum de cette liste et de cette liste privée d'un de ses maximums. Par exemple `max_reste [2; 6; 4; 6; 5]` renvoie `(6, [2; 4; 6; 5])`. On pourra procéder par correspondance de motif et traiter le cas de la liste vide par un `failwith`.

```
let rec max_reste lst =
  match lst with
  | [] -> failwith "Extraction du maximum d'une liste vide"
  | h::[] -> h, []
  | h::t -> let m,r = max_reste t in if h>m then h, m::r else m, h::r;;
```

Q9– Donner en la justifiant brièvement la complexité de la fonction `max_reste`.

Le fonction `max_reste` a une complexité linéaire en fonction de la taille n de la liste car elle effectue $n-1$ appels récursifs et chacun de ces appels n'effectue que des opérations élémentaires.

Q10– Ecrire une première version de la fonction `kmax` qui procède par extraction successive des `k` premiers maximums en utilisant la fonction `max_reste`. On procèdera de façon récursive sans utiliser les aspects impératifs de OCaml.

```
let rec kmax1 lst k =
  (*Renvoie les k premiers maximums de la liste lst ainsi que la liste lst privée
  ↪  de ces k premiers maximums*)
  if k=0 then [],lst else(
    let max, reste = max_reste lst in
    let rmax, rlst = kmax1 reste (k-1) in
    max::rmax, rlst);;
```

Q11– Quelle est la complexité de cette version de la fonction `kmax` en fonction du nombre k de maximums à extraire et de la longueur n de la liste ?

La fonction `kmax` effectue un appel récursif à `max_reste` pour chacun des k maximums à extraire et comme `max_reste` a une complexité en $\mathcal{O}(n)$, cette version de `kmax` a une complexité en $\mathcal{O}(kn)$.

■ Partie II : Résolution par un tri

Q12– Ecrire une fonction `kpremiers int list -> int -> int list` qui prend en argument une liste `lst` et un entier `k` et renvoie la liste composée des `k` premiers éléments de `lst`. Par exemple `kpremiers [2; 7; 1; 8; 5] 3` renvoie la liste `[2; 7; 1]`. On procédera de façon récursive sans utiliser les apsects impératifs de OCaml.

```
let rec kpremiers lst k =
  match lst,k with
  | _,0 -> []
  | [],_ -> failwith "Pas assez d'éléments"
  | h::t, k -> h::(kpremiers t (k-1));;
```

Q13– On propose d'écrire la fonction `kmax` en triant la liste par ordre décroissant puis en prenant ses k premiers éléments. En supposant que l'algorithme de tri utilisé a une complexité en $\mathcal{O}(n \log n)$, donner la complexité de ce nouvel algorithme (on ne demande *pas* de le programmer).

La fonction `kmax` effectue un appel à la fonction de tri qui a une complexité en $\mathcal{O}(n \log n)$ et ensuite elle effectue un appel à la fonction `kpremiers` qui a une complexité en $\mathcal{O}(k)$. Donc la complexité de cette version de `kmax` est en $\mathcal{O}(n \log n + k)$ et comme $k < n$ la complexité est en $\mathcal{O}(n \log n)$.

■ **Partie III** : Résolution en utilisant un tas

Dans la suite on suppose que la structure de données de tas d'entiers (type `int`), est *déjà implémentée* par un type `tas` sur lequel on dispose des fonctions suivantes :

- `cree_tas : int -> tas` qui prend en argument un entier `cap` et renvoie un tas binaire vide de capacité maximale `cap`.
- `donne_taille : tas -> int` qui prend en argument un tas et renvoie sa taille (le nombre d'éléments actuellement stocké dans le tas).
- `insere : int -> tas -> unit` qui insère une nouvelle valeur dans le tas. Cette fonction échoue lorsque le tas est plein.
- `donne_min : tas -> int` qui renvoie la valeur minimale contenu dans le tas *sans modifier le tas*. Cette fonction échoue lorsque le tas est vide.
- `extrait_min : tas -> int` qui renvoie, *en le supprimant du tas* le minimum du tas. Cette fonction échoue lorsque le tas est vide.

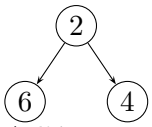
Q14– Rappeler en les justifiant, les complexités des opérations `insere` et `extrait_min` en fonction de la taille du tas notée `k` si on suppose que l'implémentation de la structure de tas est réalisée grâce à un tableau.

> Un tas est un arbre binaire complet et à chaque étape d'une insertion ou d'une extraction, on remonte (ou on descend) d'un niveau dans cet arbre, la complexité de ces opérations est donc en $O(h)$ où h est la hauteur de l'arbre or l'arbre étant complet $\mathcal{O}(h) = \mathcal{O}(\log k)$ où k est la taille de l'arbre. En conclusion, `insere` et `extrait_min` ont une complexité en $\mathcal{O}(k)$

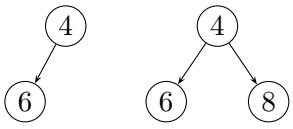
Afin d'extraire les `k` premiers éléments d'une liste de taille `n`, on propose créer un tas de taille `k` puis de parcourir récursivement la liste, pour chaque élément rencontré :

- si le tas n'est pas plein on y insère l'élément.
- sinon, on compare l'élément avec le minimum du tas, s'il est plus grand on extrait le minimum du tas et on insère l'élément dans le tas.

Par exemple, si on veut extraire les 3 premiers maximums de la liste `[4; 6; 2; 8; 3; 7; 1; 9; 5]`, après l'insertion des trois premiers éléments, le tas est :

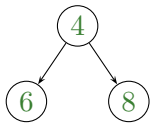


A l'étape suivante, 8 étant plus grand que 2 (le minimum du tas), on extrait 2 du tas et on y insère 8 ce qui donne :

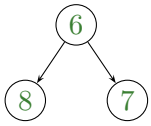


Q15– Poursuivre le déroulement de cet algorithme en faisant figurer comme ci-dessus les étapes de l'évolution du tas.

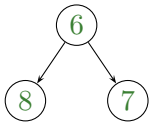
— On traite 3, comme $3 < 4$ pas de modification



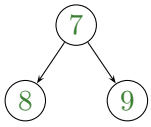
— On traite 7, comme $7 > 4$, on extrait 4 et on insère 7



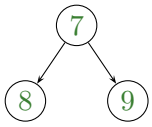
— On traite 1, comme $1 < 6$, pas de modification



— On traite 9, comme $9 > 6$, on extrait 6 et on insère 9



— On traite 5, comme $5 < 7$, pas de modification



Q16– Donner une implémentation de la fonction `kmax` utilisant ce nouvel algorithme. On rappelle qu'on pourra utiliser les fonctions de manipulation de la structure de `tas` données en début de partie. Comme précédemment, on procédera par récurrence sans utiliser les aspects impératifs de OCaml.

```
let kmax3 lst k=
  let th = cree_tas k in
  let rec aux lst =
    match lst with
    (* Si la liste est vide, tous les éléments ont été traités, les k maximums
    ↪  sont les k éléments du tas*)
    | [] -> List.init k (fun n-> extract_min th)
    (* Sinon si le tas n'est pas plein on insère l'élément rencontré*)
    | h::t -> if get_size th <k then (insert h th; aux t) else
      (* Sinon on compare l'élément rencontré avec le minimum du tas, si l'élément
      ↪  est plus grand on remplace le minimum par l'élément rencontré*)
      (* Sinon on ne fait rien et on continue à traiter la liste*)
        (let mt = get_min th in
         if h>mt then (ignore (extract_min th); insert h th; aux t) else (aux t))
  in
  aux lst;;
```

Q17– Donner en la justifiant la complexité de ce nouvel algorithme en fonction de k et n.

Les opérations d'insertion et d'extraction dans le tas sont toutes en $O(\log k)$ car le tas est de taille k. On effectue ces opérations au plus n fois (une fois pour chaque élément du tableau) et donc la complexité de ce nouvel algorithme est $O(n \log k)$

❑ **Exercice 4** : *Saut de valeur maximale* 🎓CAPES NSI 2023

Les fonctions demandées dans cet exercice sont à écrire en langage C.

Dans un tableau de flottants (type `double` du langage C) `tab` de taille `n`, on appelle *saut* un couple (i, j) avec $0 \leq i \leq j <$ `n` et la *valeur* d'un saut est la valeur `tab[j]-tab[i]`. Le but de l'exercice est de rechercher la valeur maximale d'un saut dans un tableau. Par exemple, dans le tableau `{ 2.0, 0.2, 3.0, 5.3, 2.0}`, la valeur maximale d'un saut est `5.3 - 0.2 = 5.1`, cette valeur est obtenue en considérant le saut `(1,3)`.

■ **Partie I** : Questions préliminaires et résolution naïve

Q18– Ecrire une fonction de signature `int valeur(int tab[], int i, int j, int n)` qui prend en argument un tableau `tab` de taile `n` ainsi que deux indices `i` et `j` et renvoie la valeur du saut `(i,j)`. On vérifiera les préconditions sur `i` et `j` à l'aide d'instructions `assert`. Par exemple si le tableau `tab` est `{2.0; 0.2; 3.0; 5.3; 2.0}` alors `valeur(tab, 2, 0, 5)` renvoie `1.0` (car `tab[2]-tab[0] = 1.0`).

```
int valeur(int tab[], int i, int j, int n)
{
    assert(0 <= i && i <= j && j < n);
    return tab[j] - tab[i];
}
```

Q19– Donner un exemple de tableau avec exactement deux sauts de valeur maximale et préciser ces sauts.

La liste `[2, 6, 1, 5]` possède deux sauts de valeurs maximale : `(0,1)` et `(2,3)` (ces deux sauts ont une valeur de 4)

Q20– À l'aide d'un contre-exemple, montrer qu'on ne peut pas se contenter de chercher le minimum et le maximum d'un tableau pour trouver un saut de valeur maximale.

Dans la liste `[2, 6, 1, 5]` le minimum est à l'indice 2 (c'est 1) et le maximum à l'indice 1 (c'est 3) et comme le minimum est après le maximum ce n'est pas le saut maximal.

Q21– Écrire une fonction `sautmax_naif` qui renvoie un saut de valeur maximale dans un tableau de taille `n` en testant tous les couples (i, j) tels que $0 \leq i \leq j < \texttt{n}$.

```
int valeur(int tab[], int i, int j, int n)
{
    assert(0 <= i && i <= j && j < n);
    return tab[j] - tab[i];
}
```

Q22– Quelle est la complexité de la fonction `sautmax_naif` en fonction de la taille n du tableau ?

Il y a deux boucles `for` imbriquées et les deux sont exécutés au plus `n fois`, les opérations à l'intérieur de ces boucles sont toutes des opérations élémentaires, donc la complexité est en $\mathcal{O}(n^2)$

■ **Partie II** : Résolution avec une méthode diviser pour régner

On propose maintenant d'utiliser une méthode diviser pour régner afin de calculer la valeur maximale d'un saut. On note n la taille du tableau `t` et $p = \lfloor \frac{n}{2} \rfloor$ (où $\lfloor\ \rfloor$ désigne la partie entière). On souhaite calculer :

— (i_g, j_g) un saut de valeur maximale lorsque $j_g < p$ (c'est à dire un saut maximal dans la moitié gauche)

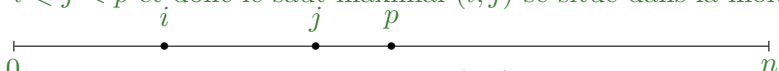
— (i_d, j_d) un saut de valeur maximale lorsque $id \geqslant p$ (c'est à dire un saut maximal dans la moitié gauche)

— (i_m, j_m) un saut de valeur maximal lorsque $i_m < p < j_m$ (c'est à dire un saut maximal dont le premier indice est dans la moitié gauche et le second dans la moitié droite)

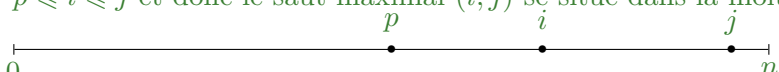
Q23– Justifier qu'un saut de valeur maximale du tableau `t` est nécessairement un des trois ci-dessus. On pourra faire un schéma pour illustrer le raisonnement.

Un saut de valeur maximal (i, j) du tableau `t` est tel que $i \leqslant j$ trois situations sont donc possibles en fonction de la position relative de i, j et p :

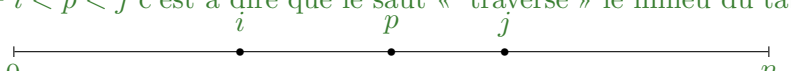
— $i \leqslant j < p$ et donc le saut maximal (i, j) se situe dans la moité gauche du tableau.



— $p \leqslant i \leqslant j$ et donc le saut maximal (i, j) se situe dans la moité droite du tableau.



— $i < p < j$ c'est à dire que le saut « traverse » le milieu du tableau.



Ce qui correspond bien aux trois situations décrites ci-dessus.

Q24– Justifier que i_m est nécessairement l'indice d'une valeur minimale dans la moitié gauche de `t` (on admettra que de même j_m est nécessairement l'indice d'une valeur maximale dans la moitié droite de `t`).

On raisonne par l'absurde, si tel n'était pas le cas, on aurait une valeur d'indice q dans la moitié gauche strictement inférieur à $t[i_m]$ et donc le saut (q, j_m) aurait une valeur supérieure au saut (i_m, j_m) ce qui contredit que (i_m, j_m) est un saut de valeur maximale.

Q25– Ecrire une fonction de signature `int min(int tab[], int a, int b)` qui prend en argument un tableau `tab`, ainsi que deux entiers `a` et `b` (avec `a<=b`) et renvoie l'indice d'un minimum de `tab` entre les deux indices `a` (inclus) et `b` (exclu).
On supposera dans la suite *déjà écrite* une fonction qui `max` qui prend les mêmes arguments et renvoie l'indice d'un maximum du sous tableau `{tab[a],...,tab[b-1]}`

```
int min(int tab[], int a, int b)
{
    assert(a < b);
    int minv = tab[a];
    for (int i = a + 1; i < b; i++)
    {
        if (tab[i] < minv)
        {
            minv = tab[i];
        }
    }
    return minv;
}
```

Q26– Ecrire une fonction de signature `int sautmax_dpr(int tab[], int a, int b)` qui prend en argument un tableau `tab` ainsi que deux entiers `a` et `b` (avec `a<=b`) et renvoie la valeur d'un saut maximale dans `tab` entre les deux indices `a` (inclus) et `b` (exclu).*Cette fonction doit être récursive et utiliser la méthode diviser pour régner.* On pourra supposer déjà écrite une fonction `max3` qui renvoie le maximum des trois `double` donnés en argument.

```c
int sautmax_dpr(int tab[], int a, int b)
{
    // Renvoie le saut maximal du sous tableau t[a],...t[b-1]
    // en utilisant une stratégie diviser pour régner
    if (b - a < 2)
    {
        return 0;
    }
    int p = (b - a) / 2;
    int smg = sautmax_dpr(tab, a, a + p);
    int smd = sautmax_dpr(tab, a + p, b);
    int ming = min(tab, a, a + p);
    int maxd = max(tab, a + p, b);
    int smax = max3(smg, smd, maxd - ming);
    return smax;
}
```

Q27– Déterminer la complexité de la fonction `sautmax_dpr`.

On obtient l'équation de compexité $C(2n) = 2C(n) + \mathcal{O}(n)$ en effet on résout deux sous problèmes de taille deux fois plus petite et on doit ensuite calculer un minimum et un maximum d'une liste de taille n et ces opérations sont en $\mathcal{O}(n)$. On suppose sans perdre de généralité que $n = 2^k$ et on note $u_k = C(2^k)/2^k$, en divisant l'équation de complexité par 2^{k+1} on obtient alors :
$u_{k+1} \leqslant u_k + \dfrac{M2^k}{2^{k+1}}$ donc, $u_{k+1} \leqslant u_k + M \times \dfrac{1}{2}$ et en sommant pour $i = 0$ à k on obtient :
$u_k \leqslant u_0 + M'k$
$C(n) \leqslant n\ (C(1) + M' \log n)$
Et donc $C(n) \in \mathcal{O}(n \log n)$.

■ **Partie III** : Résolution par programmation dynamique

On cherche maintenant à résoudre ce problème par programmation dynamique, et on adopte les notations suivantes :

- t est un tableau de taille n contenant des flottants
- $t[i]$ est l'élément d'indice i $(0 \leqslant i < n)$ de t,
- pour $0 < k \leqslant n$, t_k est le sous tableau $t[0], \ldots, t[k-1]$
- m_k est l'indice d'un minimum de t_k pour $0 < k < n$.
- (i_k, j_k) est un saut de valeur maximale dans t_k pour $0 < k < n$.

Q28– Donner les valeurs de i_1, j_1 et m_1

Comme le tableau ne contient qu'un seul élément (celui d'indice 0), $i_1 = 0, j_1 = 0$ et $m_1 = 0$.

Q29– Donner la relation de récurrence liant m_{k+1}, m_k et $t[k+1]$.

Si $t[k+1] < m_k$ alors $m_{k+1} = t[k+1]$ sinon $m_{k+1} = m_k$.

Q30– Justifier que la relation suivante est correcte :

$$(i_{k+1}, j_{k+1}) = \begin{cases} (i_k, j_k) \text{ si } t[k] - t[m_k] < t[j_k] - t[i_k] \\ (m_k, k) \text{ sinon} \end{cases}$$

Le saut de valeur maximal ayant k comme second indice est (m_k, k) et vaut $t[k] - t[m_k]$. En effet le second indice étant fixé on l'obtient en prenant le miminum du sous tableau t_k. On doit donc comparer ce nouveau saut avec le le saut maximal du sous tableau t_k. Soit il est plus petit c'est à dire $t[k] - t[m_k] < t[j_k] - t[i_k]$ et donc $(i_{k+1}, j_{k+1}) = (i_k, j_k)$ ou bien il est plus grand et donc $(i_{k+1}, j_{k+1}) = (m_k, k)$.

Q31– Ecrire une fonction de signature `int sautmax_dyn(int tab[], int n)` qui prend en argument un tableau et sa taille et renvoie la valeur maximale d'un saut de ce tableau. On procèdera *de façon ascendante* en utilisant les relations de récurrences de la question précédente et en calculant successivement les valeurs maximales de saut dans les sous tableaux t_1 puis t_2, ... jusqu'à t_n.

```c
int sautmax_dyn(int tab[], int n)
{
    int i, j, m;
    i = 0;
    j = 0;
    m = 0;
    for (int k = 1; k < n; k++)
    {
        // Mettre à jour l'indice du minimum
        if (tab[k] < tab[m])
        {
            m = k;
        }
        // Mettre à jour la valeur maximale du saut
        if (tab[k] - tab[m] >= valeur(tab, i, j, n))
        {
            i = m;
            j = k;
        }
    }
    return valeur(tab, i, j, n);
}
```

Q32– Déterminer la complexité de la fonction `sautmax_dyn`

La fonction parcourt le tableau à l'aide d'une boucle `for` qui ne contient que des opérations élémentaires, la complexité est donc linéaire en fonction de la taille du tableau.