Algorithmique avancée : introduction à la récursivité & complexité (partie 3)

Marin Bougeret

IUT Montpellier

Plan général

Partie I : algorithmes récursifs

- conception d'algorithmes récursifs (I) : exemples introductifs
- conception d'algorithmes récursifs (II) : tests, idée de preuve, exemples
- conception d'algorithmes récursifs (III) : diviser pour régner

Partie II : structures récursives (listes, arbres..)

- ..

Partie III : complexité

- ..

Introduction

Principe

- Pour l'instant, on casse une entrée de taille n en une entrée taille $n-1$, et une entrée de taille 1
- Principe du "diviser pour regner" :
  casser une entrée de taille n en deux entrées de taille $\frac{n}{2}$
  (et faire deux appels récursifs)
  ou plus généralement,
  casser une entrée de taille n en c entrées de taille $\approx \frac{n}{c}$
  (et faire c appels récursifs)

Introduction

Principe

- Pour l'instant, on casse une entrée de taille n en une entrée taille $n-1$, et une entrée de taille 1
- Principe du "diviser pour regner" :
  casser une entrée de taille n en deux entrées de taille $\frac{n}{2}$
  (et faire deux appels récursifs)
  ou plus généralement,
  casser une entrée de taille n en c entrées de taille $\approx \frac{n}{c}$
  (et faire c appels récursifs)

Introduction

Principe

- Pour l'instant, on casse une entrée de taille n en une entrée taille $n-1$, et une entrée de taille 1
- Principe du "diviser pour regner" :
  casser une entrée de taille n en deux entrées de taille $\frac{n}{2}$
  (et faire deux appels récursifs)
  ou plus généralement,
  casser une entrée de taille n en c entrées de taille $\approx \frac{n}{c}$
  (et faire c appels récursifs)

Pourquoi cela ?

- pourquoi pas! la récursivité n'est pas limitée à faire un appel sur une entrée de taille $n-1$
- pour des raisons de temps d'exécution (complexité)

Principe

- Pour l'instant, on casse une entrée de taille n en une entrée taille $n-1$, et une entrée de taille 1
- Principe du "diviser pour regner" :
  casser une entrée de taille n en deux entrées de taille $\frac{n}{2}$
  (et faire deux appels récursifs)
  ou plus généralement,
  casser une entrée de taille n en c entrées de taille $\approx \frac{n}{c}$
  (et faire c appels récursifs)

Pourquoi cela ?

- pourquoi pas! la récursivité n'est pas limitée à faire un appel sur une entrée de taille $n-1$
- pour des raisons de temps d'exécution (complexité)

Principe

- Pour l'instant, on casse une entrée de taille n en une entrée taille $n-1$, et une entrée de taille 1
- Principe du "diviser pour regner" :
  casser une entrée de taille n en deux entrées de taille $\frac{n}{2}$
  (et faire deux appels récursifs)
  ou plus généralement,
  casser une entrée de taille n en c entrées de taille $\approx \frac{n}{c}$
  (et faire c appels récursifs)

Pourquoi cela ?

- pourquoi pas! la récursivité n'est pas limitée à faire un appel sur une entrée de taille $n-1$
- pour des raisons de temps d'exécution (complexité)

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechercheAux(int[] t, int x, int i){
  // 0 <= i <= t.length, t trie
  // ret. vrai ssi x dans t[i..(t.length-1)]
  if( i == t.length ){return false;}
  if( t[i] > x){return false;}
  else{
    return ((t[i]==x)||rechercheAux(t,x,i+1));
  }
}
```

But

- écrire un algorithme beaucoup plus rapide que rechercheAux

Comment mesurer le temps d'exécution

- que compter : le nb. d'opérations élémentaires (+,*,==,=,..)
- sur quelle entrée : on compte le nombre d'opérations dans le pire des cas

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechercheAux(int[] t, int x, int i){
  // 0 <= i <= t.length, t trie
  // ret. vrai ssi x dans t[i..(t.length-1)]
  if( i == t.length ){return false;}
  if( t[i] > x){return false;}
  else{
    return ((t[i]==x)||rechercheAux(t,x,i+1));
  }
}
```

But

- écrire un algorithme beaucoup plus rapide que rechercheAux

Comment mesurer le temps d'exécution

- que compter : le nb. d'opérations élémentaires (+,*,==,=,..)
- sur quelle entrée : on compte le nombre d'opérations dans le pire des cas

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechercheAux(int[] t, int x, int i){
  // 0 <= i <= t.length, t trie
  // ret. vrai ssi x dans t[i..(t.length-1)]
  if( i == t.length ){return false;}
  if( t[i] > x){return false;}
  else{
    return ((t[i]==x)||rechercheAux(t,x,i+1));
  }
}
```

But

- écrire un algorithme beaucoup plus rapide que rechercheAux

Comment mesurer le temps d'exécution

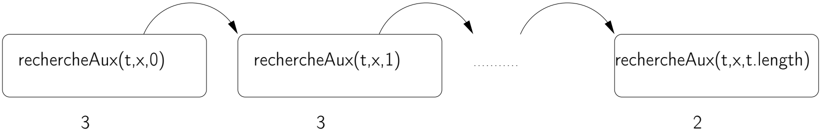
- que compter : le nb. d'opérations élémentaires (+,*,==,=,..)
- sur quelle entrée : on compte le nombre d'opérations dans le pire des cas

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechercheAux(int[] t, int x, int i){
  // 0 <= i <= t.length, t trie
  // ret. vrai ssi x dans t[i..(t.length-1)]
  if( i == t.length ){return false;}
  if( t[i] > x){return false;}
  else{
    return ((t[i]==x)||rechercheAux(t,x,i+1));
  }
}
```

Nombre d'opérations de rechercheAux(t,x,0) dans le pire des cas

$\approx 3n$ ($n = t.length$)



Exemple : améliorer rechercheAux(int[] t, int x, int i)

Appliquons maintenant une stratégie diviser pour regner

Pour chercher x dans un tableau trié :

- on regarde si x est au milieu ($x == t[\frac{t.length}{2}]$)
- si x est plus petit : on recherche récursivement x dans le "demi tableau" gauche
- si x est plus grand : on recherche récursivement x dans le "demi tableau" droite

On utilise l'astuce pour éviter les recopies de sous tableaux :

```
boolean rechDicho(int []t, int x, int i, int j){
 // t trie (vide eventuellement)
 // 0 <= i
 // j <= t.length-1
 // ret. vrai ssi x dans t[i..j]
```

Exemple : améliorer rechercheAux(int[] t, int x, int i)

Appliquons maintenant une stratégie diviser pour regner

Pour chercher x dans un tableau trié :

- on regarde si x est au milieu ($x == t[\frac{t.length}{2}]$)
- si x est plus petit : on recherche récursivement x dans le "demi tableau" gauche
- si x est plus grand : on recherche récursivement x dans le "demi tableau" droite

On utilise l'astuce pour éviter les recopies de sous tableaux :

```
boolean rechDicho(int []t, int x, int i, int j){
 // t trie (vide eventuellement)
 // 0 <= i
 // j <= t.length-1
 // ret. vrai ssi x dans t[i..j]
```

Exemple : améliorer rechercheAux(int[] t, int x, int i)

Appliquons maintenant une stratégie diviser pour regner

Pour chercher x dans un tableau trié :

- on regarde si x est au milieu ($x == t[\frac{t.length}{2}]$)
- si x est plus petit : on recherche récursivement x dans le "demi tableau" gauche
- si x est plus grand : on recherche récursivement x dans le "demi tableau" droite

On utilise l'astuce pour éviter les recopies de sous tableaux :

```
boolean rechDicho(int []t, int x, int i, int j){
 // t trie (vide eventuellement)
 // 0 <= i
 // j <= t.length-1
 // ret. vrai ssi x dans t[i..j]
```

Exemple : améliorer rechercheAux(int[] t, int x, int i)

Appliquons maintenant une stratégie diviser pour regner

Pour chercher x dans un tableau trié :

- on regarde si x est au milieu ($x == t[\frac{t.length}{2}]$)
- si x est plus petit : on recherche récursivement x dans le "demi tableau" gauche
- si x est plus grand : on recherche récursivement x dans le "demi tableau" droite

On utilise l'astuce pour éviter les recopies de sous tableaux :

```
boolean rechDicho(int []t, int x, int i, int j){
 // t trie (vide eventuellement)
 // 0 <= i
 // j <= t.length-1
 // ret. vrai ssi x dans t[i..j]
```

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechDicho(int[] t, int x, int i, int j)
  // 0<=i et j<=t.length-1, (on impose pas i<=j)
  // t trie
  // ret. vrai ssi x dans t[i..j]


    int m = (i+j)/2;
    if(x==t[m]){return true;}
    else if(x<t[m])
       return rechDicho(t,x,i,m-1);
         else
       return rechDicho(t,x,m+1,j);
```

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechDicho(int[] t, int x, int i, int j)
  // 0<=i et j<=t.length-1, (on impose pas i<=j)
  // t trie
  // ret. vrai ssi x dans t[i..j]


    int m = (i+j)/2;
    if(x==t[m]){return true;}
    else if(x<t[m])
       return rechDicho(t,x,i,m-1);
         else
       return rechDicho(t,x,m+1,j);
```

$x \in E$ qui provoquent une erreur dans I):

- $j < 0$ .. pas clair (si $j == -1$ et $i == t.length$, t[m] peut ne pas poser problème. Et l'appel rec ?)
- $i \geq t.length$ .. pareil

Conclusion : il est pénible de déterminer exactement les entrées qui provoquent une erreur dans I)

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechDicho(int[] t, int x, int i, int j)
  // 0<=i et j<=t.length-1, (on impose pas i<=j)
  // t trie
  // ret. vrai ssi x dans t[i..j]


    int m = (i+j)/2;
    if(x==t[m]){return true;}
    else if(x<t[m])
       return rechDicho(t,x,i,m-1);
         else
       return rechDicho(t,x,m+1,j);
```

- Idée : on détermine un sur-ensemble des entrées qui provoquent une erreur ..
- .. et on ajoute un cas de base pour toutes ces entrées
- autrement dit, on ajoute peut être trop de cas de base, mais ça n'est pas grave!

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechDicho(int[] t, int x, int i, int j)
  // 0<=i et j<=t.length-1, (on impose pas i<=j)
  // t trie
  // ret. vrai ssi x dans t[i..j]


    int m = (i+j)/2;
    if(x==t[m]){return true;}
    else if(x<t[m])
       return rechDicho(t,x,i,m-1);
         else
       return rechDicho(t,x,m+1,j);
```

- ici, $i > j$ semble contenir tous les cas dangereux ..
- .. et pour $i > j$, on sait répondre
- on ajoute donc un cas de base

Exemple : améliorer rechercheAux(int[] t, int x, int i)

```
boolean rechDicho(int[] t, int x, int i, int j)
  // 0<=i et j<=t.length-1, (on impose pas i<=j)
  // t trie
  // ret. vrai ssi x dans t[i..j]

    if(i>j) return false;
    int m = (i+j)/2;
    if(x==t[m]){return true;}
    else if(x<t[m])
       return rechDicho(t,x,i,m-1);
         else
       return rechDicho(t,x,m+1,j);
```

- est-ce suffisant ?
- est ce que $i > j$ contient tous les cas provoquant une erreur ?

$\Leftrightarrow$ est ce que pour tout $i \leq j$ il n'y a jamais d'erreur dans I) ?

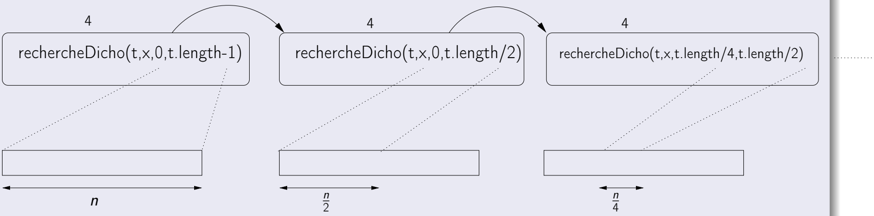
- on vérifie : tout va bien!

Exemple : améliorer rechercheAux(int[] t, int x, int i)

(A ne pas faire d'habitude) :
Executons rechercheAux([1,3,7,10,14,15,18],7,0,6) à la min

Exemple : améliorer rechercheAux(int[] t, int x, int i)

Nb d'op de rechercheDicho(t,x,0,t.length-1) dans le pire des cas

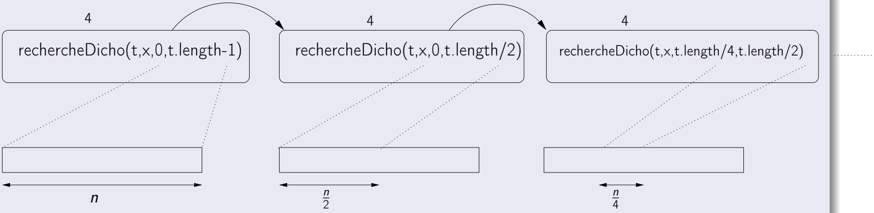


- il y a $log(n)$ étapes car :
    - après x étapes, le sous tableau est de taille $\frac{n}{2^x}$

    $\Rightarrow$ après $log(n)$ étapes, sous tableau de taille 1
- il y a ≤ 4 opérations dans chaque étape
- donc au total $\approx 4log(n)$ opérations

Exemple : améliorer rechercheAux(int[] t, int x, int i)

Nb d'op de rechercheDicho(t,x,0,t.length-1) dans le pire des cas

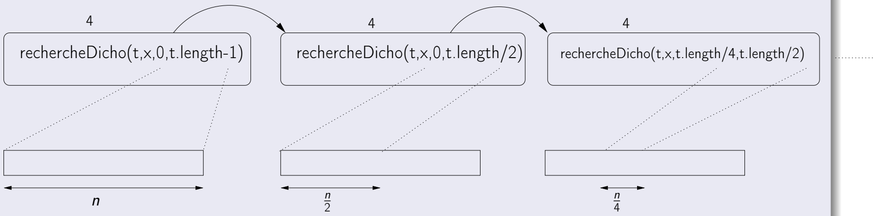


- il y a $log(n)$ étapes car :
  - après x étapes, le sous tableau est de taille $\frac{n}{2^x}$

  $\Rightarrow$ après $log(n)$ étapes, sous tableau de taille 1
- il y a ≤ 4 opérations dans chaque étape
- donc au total $\approx 4log(n)$ opérations

Exemple : améliorer rechercheAux(int[] t, int x, int i)

Nb d'op de rechercheDicho(t,x,0,t.length-1) dans le pire des cas

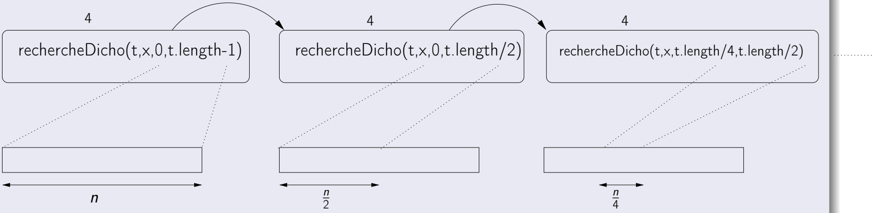


- il y a $log(n)$ étapes car :
  - après x étapes, le sous tableau est de taille $\frac{n}{2^x}$

  $\Rightarrow$ après $log(n)$ étapes, sous tableau de taille 1
- il y a ≤ 4 opérations dans chaque étape
- donc au total $\approx 4log(n)$ opérations

Exemple : améliorer rechercheAux(int[] t, int x, int i)

Nb d'op de rechercheDicho(t,x,0,t.length-1) dans le pire des cas



- il y a $log(n)$ étapes car :
  - après x étapes, le sous tableau est de taille $\frac{n}{2^x}$

  $\Rightarrow$ après $log(n)$ étapes, sous tableau de taille 1
- il y a ≤ 4 opérations dans chaque étape
- donc au total $\approx 4log(n)$ opérations

n Vs $log(n)$

Ne sous estimez pas notre ami log

Pour $n = 10^{80}$ (nombre estimé de particules dans l'univers)

- rechercheAux nécéssite .. 10^{80} opérations
- rechercheDichotomique nécéssite .. 265 opérations

n Vs $log(n)$

Ne sous estimez pas notre ami log

Pour $n = 10^{80}$ (nombre estimé de particules dans l'univers)

- rechercheAux nécéssite .. 10^{80} opérations
- rechercheDichotomique nécéssite .. 265 opérations

Bilan

- Principe du "diviser pour regner" :
  casser une entrée de taille n en deux entrées de taille $\frac{n}{2}$
  (et faire deux appels récursifs)
  ou plus généralement,
  casser une entrée de taille n en c entrées de taille $\approx \frac{n}{c}$
  (et faire c appels récursifs)
- Intérêt principal : écrire des algorithms beaucoup plus rapides

Remarque

Le "diviser pour regner" rentre dans notre cadre de travail habituel (méthode I) et II))

Bilan

- Principe du "diviser pour regner" :
  casser une entrée de taille n en deux entrées de taille $\frac{n}{2}$
  (et faire deux appels récursifs)
  ou plus généralement,
  casser une entrée de taille n en c entrées de taille $\approx \frac{n}{c}$
  (et faire c appels récursifs)
- Intérêt principal : écrire des algorithms beaucoup plus rapides

Remarque

Le "diviser pour regner" rentre dans notre cadre de travail habituel (méthode I) et II))

Méthode pour concevoir un algorithme récursif A

$I)$ écrire A en supposant que A fonctionne déjà pour des entrées plus petites :

- penser à une grande entrée x
- supposer que pour tout $x' \in E$ plus petite que x, $A(x')$ donnera la bonne réponse
- en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x

$II)$ ajouter des cas de base:

- ajouter des cas de base pour traiter les entrées de E provoquant une erreur dans le code $I)$

$x \in E$ provoque une erreur ds $I)$ si au moins une des cond. est v.

1. il y a une instruction incorrecte (division par 0, sortie tab...)
2. il y a un appel recursif $A(x')$ incorrect (x' ne vérifie pas les prérequis ($x' \notin E$), ou x' n'est pas plus petite que x)
3. tous les appels récursifs sont corrects, mais le calcul pour en déduire le résultat pour x est faux

Plan de la fin du cours

conception d'algorithmes récursifs (II) : divisier pour regner

- exemple 2 : tri fusion
- exemple 3 : meilleure sous somme

Exemple 2 : triFusion

Appliquons maintenant une stratégie diviser pour regner

Pour trier un tableau selon le triFusion :

- on trie la moitié gauche
- on trie la moitié droite
- on fusionne les deux moitiés triées

On utilise l'astuce pour éviter les recopies de sous tableaux : `void triFusion(int []t, int i, int j){`

```
// i,j .. à venir
//trie t[i..j] par ordre croissant
```

Exemple 2 : triFusion

Appliquons maintenant une stratégie diviser pour regner

Pour trier un tableau selon le triFusion :

- on trie la moitié gauche
- on trie la moitié droite
- on fusionne les deux moitiés triées

On utilise l'astuce pour éviter les recopies de sous tableaux : `void triFusion(int []t, int i, int j){`

```
// i,j ..  à venir
//trie t[i..j] par ordre croissant
```

Exemple 2 : triFusion

```java
void triFusion(int[] t, int i, int j){
  //0 <= i et j <= t.length-1
  //trie t[i..j] par ordre croissant


    int m=(i+j)/2;
    triFusion(t,i,m);
    triFusion(t,m+1,j);
    fusion(t,i,m+1,j);

}
```

Il restera à écrire :

```java
void fusion(int[] t,int deb1,int deb2,int fin)
  //deb1 < deb2 <= fin
  //t trie croissant entre deb1 .. deb2-1
  //t trie croissant entre deb2 .. fin
  //but : trie t[deb1..fin] croissant
```

Exemple 2 : triFusion

```
void triFusion(int[] t, int i, int j){
  //0 <= i et j <= t.length-1
  //trie t[i..j] par ordre croissant


     int m=(i+j)/2;
     triFusion(t,i,m);
     triFusion(t,m+1,j);
     fusion(t,i,m+1,j);

}
```

On détermine un sur-ensemble des $x \in E$ provoquant une erreur dans I)

- ici, $i > j$ semble contenir tous les cas dangereux ..
- .. et pour $i > j$, on sait répondre
- on ajoute donc un cas de base

```
void triFusion(int[] t, int i, int j){
  //0 <= i et j <= t.length-1
  //trie t[i..j] par ordre croissant
    if(i>j){}
    else
     int m=(i+j)/2;
     triFusion(t,i,m);
     triFusion(t,m+1,j);
     fusion(t,i,m+1,j);

}
```

- est-ce suffisant ?
- est ce que $i > j$ contient tous les cas provoquant une erreur ?

$\Leftrightarrow$ est ce que pour tout $i \leq j$ il n'y a jamais d'erreur dans I) ?

- on vérifie .. et non! pour $i = j$ il y a un appel réc pas plus petit

Exemple 2 : triFusion

```
void triFusion(int[] t, int i, int j){
  //0 <= i et j <= t.length-1
  //trie t[i..j] par ordre croissant
    if(i>=j){}
    else
     int m=(i+j)/2;
     triFusion(t,i,m);
     triFusion(t,m+1,j);
     fusion(t,i,m+1,j);

}
```

- on élargit le cas de base à $i \geq j$: est-ce suffisant ?
- est ce que $i \geq j$ contient tous les cas provoquant une erreur ?

$\Leftrightarrow$ est ce que pour tout $i < j$ il n'y a jamais d'erreur dans I) ?

- on vérifie .. tout va bien!
- en ré-écrivant le premier if "dans l'autre sens" on obtient:

Exemple 2 : triFusion

```
void triFusion(int[] t, int i, int j){
  //0 <= i et j <= t.length-1
  //trie t[i..j] par ordre croissant
  if(i<j){
    int m=(i+j)/2;
    triFusion(t,i,m);
    triFusion(t,m+1,j);
    fusion(t,i,m+1,j);
  }
}
```

Il reste à écrire :

```
void fusion(int[] t,int deb1,int deb2,int fin)
  //deb1 < deb2 <= fin
  //t trie croissant entre deb1 .. deb2-1
  //t trie croissant entre deb2 .. fin
  //but : trie t[deb1..fin] croissant
```

Exemple 2 : triFusion

```
void fusion(int[] t,int deb1,int deb2,int fin){
  int[] temp = new int[fin-deb1+1];
  int l1 = deb1; //indice 1 de prochaine lecture
  int l2 = deb2; //indice 2 de prochaine lecture
  int e = 0; //indice prochaine ecriture









}
```

Schéma tableau : fusion avec les l_i qui avancent

Exemple 2 : triFusion

```
void fusion(int[] t,int deb1,int deb2,int fin){
  int[] temp = new int[fin-deb1+1];
  int l1 = deb1; //indice 1 de prochaine lecture
  int l2 = deb2; //indice 2 de prochaine lecture
  int e = 0; //indice prochaine ecriture
  while(e < temp.length){
    if(l1==deb2)
       ..
    else if(l2==fin+1)
       ..
    else{
      if(t[l1]<t[l2])
        temp[e]=t[l1];l1++;
      else
        ..
    }
    e++;
  }
  for .. //recopie temp dans t[deb1..fin]
}
```

Exemple 3 : meilleure sous somme

Objectif

Etant donné un tableau t contenant des entiers, trouver la plus grande somme (eventuellement vide) de cases consécutives de t.

5	-6	4	-1	2	-5	1

5

Exemple 3 : meilleure sous somme

Version 1 : sans diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme dans les $n-1$ dernières cases
- on calcule la meilleure sous somme qui contient la première case
- on retourne la plus grande des deux valeurs

Exemple 3 : meilleure sous somme

Version 1 : sans diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme dans les $n-1$ dernières cases
- on calcule la meilleure sous somme qui contient la première case
- on retourne la plus grande des deux valeurs

Exemple 3 : meilleure sous somme

Version 1 : sans diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme dans les $n-1$ dernières cases
- on calcule la meilleure sous somme qui contient la première case
- on retourne la plus grande des deux valeurs

Exemple 3 : meilleure sous somme

```
public int bestSousSomme(int[] t, int i){
  //0 <= i <= t.length
  //calcule meilleure SS du sous tableau
  //t[i..(t.length-1)]
  if(i==t.length)
    return 0;
  else{
    int a = bestSousSomme(t,i+1);
    int b = Aux(t,i); //calcule la meilleure
        SS qui commence par t[i]
    return max(a,b);
  }
}
```

Exemple 3 : meilleure sous somme

Version 2 : avec diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme incluse dans la moitié gauche
- on calcule récursivement la meilleure sous somme incluse dans la moitié droite
- on calcule la meilleure sous somme qui contient la case du milieu
- on retourne la plus grande des trois valeurs

Exemple 3 : meilleure sous somme

Version 2 : avec diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme incluse dans la moitié gauche
- on calcule récursivement la meilleure sous somme incluse dans la moitié droite
- on calcule la meilleure sous somme qui contient la case du milieu
- on retourne la plus grande des trois valeurs

Exemple 3 : meilleure sous somme

Version 2 : avec diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme incluse dans la moitié gauche
- on calcule récursivement la meilleure sous somme incluse dans la moitié droite
- on calcule la meilleure sous somme qui contient la case du milieu
- on retourne la plus grande des trois valeurs

Exemple 3 : meilleure sous somme

Version 2 : avec diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme incluse dans la moitié gauche
- on calcule récursivement la meilleure sous somme incluse dans la moitié droite
- on calcule la meilleure sous somme qui contient la case du milieu
- on retourne la plus grande des trois valeurs

Exemple 3 : meilleure sous somme

Version 2 : avec diviser pour régner

Pour chercher la meilleure sous somme (cf schéma) :

- on calcule récursivement la meilleure sous somme incluse dans la moitié gauche
- on calcule récursivement la meilleure sous somme incluse dans la moitié droite
- on calcule la meilleure sous somme qui contient la case du milieu
- on retourne la plus grande des trois valeurs

Exemple 3 : meilleure sous somme

```
int bestSSDivide(int[] t, int i, int j){
  //t (eventuellement vide)
  //0 <= i
  // j <= t.length-1
  //calcule la meilleure sous somme de t[i..j]
  if(i>j){
    return 0;
  }
  else{ // i <= j
    int m=(i+j)/2;
    int a = bestSSDivide(t,i,m-1);
    int b = bestSSDivide(t,m+1,j);
    int c = Aux2(t,i,m,j);//retourne meilleure
       SS contenant t[m]
    return max(a,b,c);
  }
}
```

Exemple 3 : meilleure sous somme

- bravo moi ! j'ai écrit meilleure SS. en "diviser pour régner"
- mais .. est-ce plus rapide ?

Calcul du nombre d'opérations dans le pire cas

bestSousSomme (version 1)

- coût sur un tableau de taille n :
  n opérations + un appel récursif de taille $n-1$
- $f(n) = n + f(n-1)$
- $f(n) \approx n^2$

bestSSDivide (version 2)

- coût sur un tableau de taille n :
  n opérations + deux appels récursifs de taille $\frac{n}{2}$
- $g(n) = n + 2g(\frac{n}{2})$
- $g(n) \approx nlog(n)$ (il existe une solution en $\approx n$ (cf TD))

Exemple 3 : meilleure sous somme

- bravo moi ! j'ai écrit meilleure SS. en "diviser pour régner"
- mais .. est-ce plus rapide ?

Calcul du nombre d'opérations dans le pire cas

bestSousSomme (version 1)

- coût sur un tableau de taille n :
  n opérations + un appel récursif de taille $n-1$
- $f(n) = n + f(n-1)$
- $f(n) \approx n^2$

bestSSDivide (version 2)

- coût sur un tableau de taille n :
  n opérations + deux appels récursifs de taille $\frac{n}{2}$
- $g(n) = n + 2g(\frac{n}{2})$
- $g(n) \approx nlog(n)$ (il existe une solution en $\approx n$ (cf TD))

Exemple 3 : meilleure sous somme

- bravo moi ! j'ai écrit meilleure SS. en "diviser pour régner"
- mais .. est-ce plus rapide ?

Calcul du nombre d'opérations dans le pire cas

bestSousSomme (version 1)

- coût sur un tableau de taille n :
  n opérations + un appel récursif de taille $n-1$
- $f(n) = n + f(n-1)$
- $f(n) \approx n^2$

bestSSDivide (version 2)

- coût sur un tableau de taille n :
  n opérations + deux appels récursifs de taille $\frac{n}{2}$
- $g(n) = n + 2g(\frac{n}{2})$
- $g(n) \approx nlog(n)$ (il existe une solution en $\approx n$ (cf TD))

Exemple 3 : meilleure sous somme

- bravo moi ! j'ai écrit meilleure SS. en "diviser pour régner"
- mais .. est-ce plus rapide ?

Calcul du nombre d'opérations dans le pire cas

bestSousSomme (version 1)

- coût sur un tableau de taille n :
  n opérations + un appel récursif de taille $n-1$
- $f(n) = n + f(n-1)$
- $f(n) \approx n^2$

bestSSDivide (version 2)

- coût sur un tableau de taille n :
  n opérations + deux appels récursifs de taille $\frac{n}{2}$
- $g(n) = n + 2g(\frac{n}{2})$
- $g(n) \approx nlog(n)$ (il existe une solution en $\approx n$ (cf TD))