Développement efficace (R3.02)
Récursivité II : diviser pour régner
(divide and conquer)

Marin Bougeret
LIRMM, IUT/Université de Montpellier

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)

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)

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)

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)

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

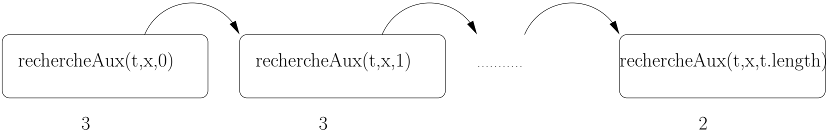
- que compter : le nb. d'opérations élémentaires (+,*,==,=,..)
- sur quelle entrée : on compte le nombre d'opérations dans le pire des cas

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

Cherchons x comme on cherche un mot dans le dictionnaire !
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

Cherchons x comme on cherche un mot dans le dictionnaire !

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

Cherchons x comme on cherche un mot dans le dictionnaire !
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

Cherchons x comme on cherche un mot dans le dictionnaire !
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

Cherchons x comme on cherche un mot dans le dictionnaire !
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

Cherchons x comme on cherche un mot dans le dictionnaire !
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);
```

Comment trouver nos 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]

    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);
```

Rappel : pour les $x \in E$ traités par récurrence, il faut s'assurer que

1. toutes les instructions sont correctes (pas de division par 0, sortie de tableau, ..)
2. les appels récursifs $A(x')$ sont corrects (x' vérifie les prérequis ($x' \in E$), et x' plus petit que x)
3. le calcul qui déduit le résultat des appels récursifs est correct

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);
```

Rappel : une technique : écrire d'abord les cas traités par récurrence (en pensant à des x très grands), voir pour quels x elle n'est pas correct, et ajouter des cas de base pour ceux là

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, pas facile de déterminer exactement les entrées où la récurrence n'est pas correct :
- $j = t.length + 10$, $j = -2$ correct ?

$\rightarrow$ on va plutôt proposer un cas de base facile à éxprimer et pour lequel on sait résoudre facilement .. et voir si cela suffit!

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 le bloc de récurrence ?

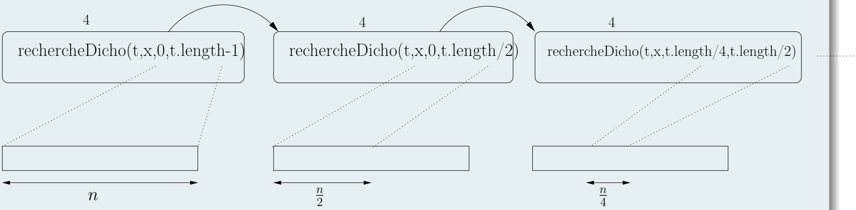
- on vérifie : tout va bien!

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 main

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

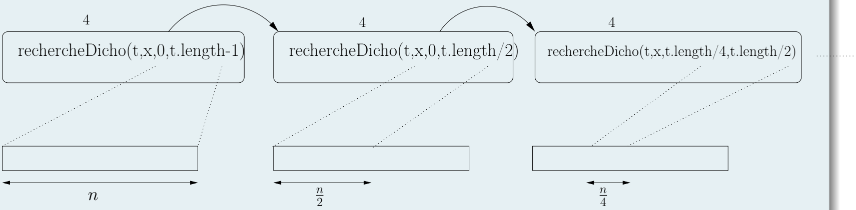


- il y a $log(n)$ étapes car :
  - après x étapes, le sous tableau est de taille $\frac{n}{2^x}$

  $\Rightarrow$ 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

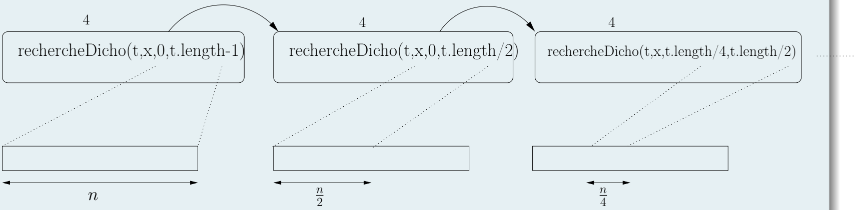


- il y a $log(n)$ étapes car :
  - après x étapes, le sous tableau est de taille $\frac{n}{2^x}$

  $\Rightarrow$ 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

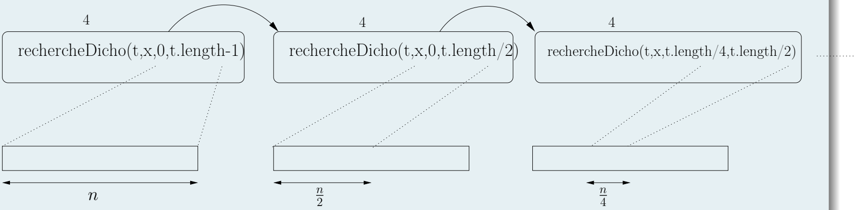


- il y a $log(n)$ étapes car :
  - après x étapes, le sous tableau est de taille $\frac{n}{2^x}$

  $\Rightarrow$ après $log(n)$ étapes, sous tableau de taille 1
- il y a $\leq$ 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

- 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

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

- 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

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

```
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 :

```
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
  if(i>=j){}
  else
     int m=(i+j)/2;
     triFusion(t,i,m);
     triFusion(t,m+1,j);
     fusion(t,i,m+1,j);
}
```

Ajout un cas de base raisonnable. Si $i < j$, est ce que le bloc de récurrence est ok ?

- instructions correctes : ok
- appels légaux ok :
  - `triFusion(t,i,m)` ok car $0 <= i$ et $m <= t.length - 1)$
  - `triFusion(t,m+1,j)` et `fusion(t,i,m+1,j)` ok aussi

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);
}
```

Ajout un cas de base raisonnable. Si $i < j$, est ce que le bloc de récurrence est ok ?

- instructions correctes : ok
- paramètre x' plus petit ?
  - `triFusion(t,i,m)` : (t, i, m) plus petit que (t, i, j) ok
  - `triFusion(t,m+1,j)` : $(t, m+1, j)$ plus petit que (t, i, j) ok

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);
}
```

En ré-ecrivant le "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]
}
```

Conclusion

- La technique de diviser pour régner (divide and conquer) est une technique algorithmique utilisée dans de très nombreux algorithmes du folkore
  - recherche dichotomique,
  - tri fusion,
  - algorithme de Karatsuba (pour mutliplier deux nombres de n chiffres chacun en faisant moins de n^2 mutliplications élémentaires)
  - ..

Complexité

- Diviser pour régner permet d'écrire des algorithmes plus "rapides" (faisant moins d'opérations, on dit aussi "de meilleure complexité").
- Discutons quelques instants du nombre d'opérations d'un algorithme récursif qui fait plusieurs appels récursifs

On considère la suite u_n définie par $u_1 = 1$, $u_n = u_{n-1}^2 + (n+3)$

V1

```
int u(int n){
  if(n==1) // cas de base
    return 1;
   else{
    int temp = u(n-1);//  temp = u_{n-1}
    return temp*temp+(n+3);
  }
}
```

Complexité : $\mathcal{O}(n)$. Preuve :

On considère la suite u_n définie par $u_1 = 1$, $u_n = u_{n-1}^2 + (n+3)$

V2

```
int u(int n){
  if(n==1) // cas de base
    return 1;
   else{
    return u(n-1)*u(n-1)+(n+3);
  }
}
```

Complexité : $\mathcal{O}(2^n)$. Preuve :

Complexité d'algorithmes récursifs

Morale : deux appels récursifs d'ordre $(n-1)$: très grosse complexité (typiquement 2^n).

Et dans le diviser pour régner, quid de deux appels récursifs d'ordre $n/2$ par exemple (comme dans triFusion) ?

Complexité d'algorithmes diviser pour régner

- le nombre d'opérations d'un algorithme de divide and conquer s'exprimera typiquement sous la forme $T(n) \leq aT(\frac{n}{b}) + f(n)$
- il existe un théorème (le "master Theorem") qui selont les valeurs de a, b, f nous donne la forme explicite de $T(n)$
- pour triFusion, $a = b = 2$, et $f(n) = n$, alors $T(n) = \mathcal{O}(nlog(n))$