Cours 3

Chapitre 2 - Programmation des formules récursives : Redondance, Mémoïsation et localité

- **Programmation efficace de schémas récursifs**
  - Technique de **memoisation** pour éliminer redondance
    - Programmation foctionnelle (map, reduce, memoize)
  - Technique de **blocking** pour la localité (cache)

- **Introduction à la programmation dynamique**
  - Optimisation discrète (exemple)
  - Caractérisation récursive d'une solution optimale

Formulations récursives

- Une définition d’un objet est récursive si elle fait appel à la définition d’un objet du même type.

- Intéressant pour l’optimisation discrète:
  - Énumération récursive des solutions à envisager
  - Exemple

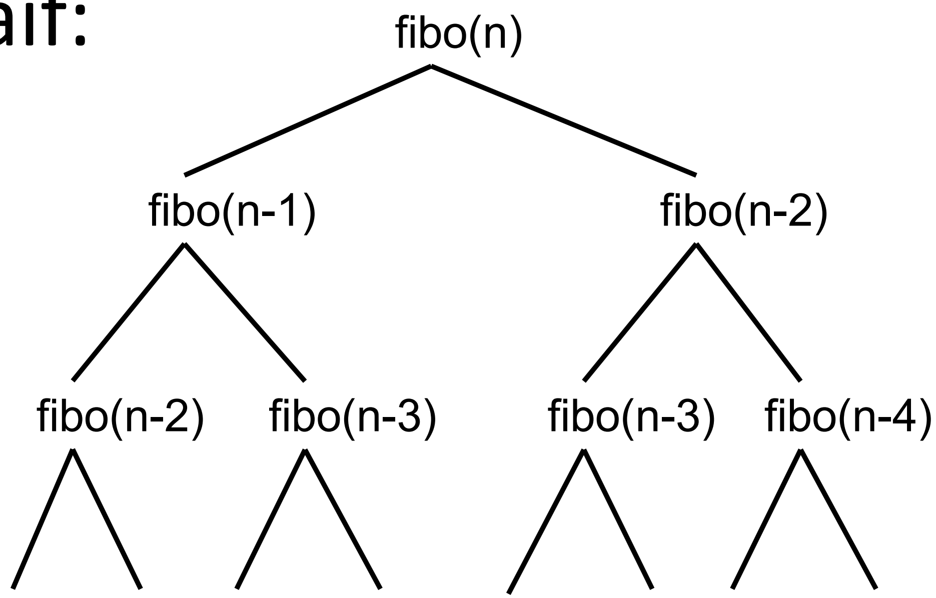
- Gloire… et déboires!
  - Algorithmes de bonne qualité (ex. D&C)
  - Ou complètement inefficaces si programmation naïve…

Exemple: suite de Fibonacci

fibo(n) = fibo(n-1) + fibo(n-2)

- Programme récursif naïf:

```
def fib(n):
   if n == 0:
      return 0
   elif n == 1:
      return 1
   else:
    return fib(n-1) + fib(n-2)
```

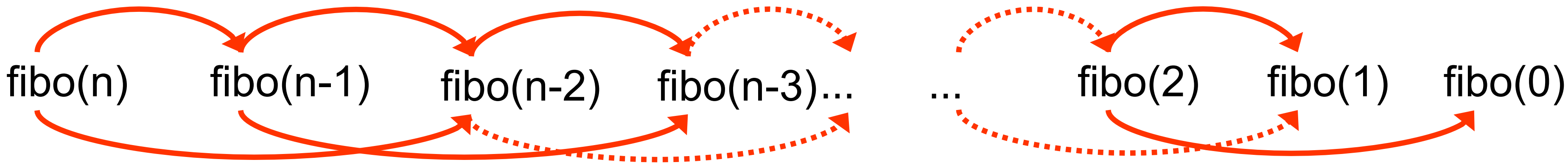


arbre d'appels

- Fib(20) =
- Fib(30) =
- Fib(40) =

Elimination des appels redondants

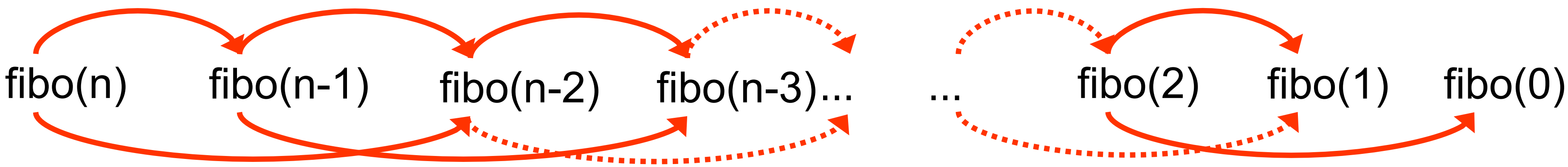
- Au lieu de représenter un arbre d'appel, on représente un **graphe d'appels**, en confondant les sommets correspondant à un même appel avec les mêmes valeurs



- Redondances bien visibles!
- Deux techniques d'élimination:
  - – soit **ordonnancement** selon ordre topologique
  - – soit « **mémoïsation** » (marquage)

Ordonnancement des calculs - Exemple: Fibonacci

- Recherche d’un ordre topologique dans le graphe des appels



- Principe de l’algorithme?
  - cours graphes: trouver un ordre de traitement des sommets (tri topologique des tâches à accomplir)

Exemple: Fibonacci

```
integer fibo (n: integer) is
    k, fib_k, fib_k_1: integer
    k := 1
    fib_k_1 := 0
    fib_k := 1
    Pour k de 2 à n faire
    aux := fib_k_1 + fib_k
    fib_k_1 := fib_k
    fib_k := aux
    Retouner fib_k
```

– On retrouve l'algorithme itératif classique!

– En fait ce raisonnement amène d'un problème récursif à une écriture itérative!

Généralisation: Mémoisation

- Mémoriser les appels effectués
  - Dans une table de hachage :
    - Clef = paramètres d’appel de la fonction
    - Valeur = valeur de l’appel avec ces paramètres
- De manière systématique
  - Automatisable dans les langages fonctionnels

Exemple: Fibonacci

- Mémoïsation:
  Algorithme récursif avec marquage

```
integer fibo (n: integer) is
        Si (tab[n] = -1) Alors -- pas encore calculé
        tab[n] := fibo(n-1) + fibo(n-2)
        Retourner tab[n]
```

tab[0]:=0
tab[1]:=1

Inconvénient?   O(n) en mémoire!

Mémorisation du résultat des appels

```
def fib_memo_basique(n):
    memo = {} #table de memorisation du résultat des appels
    def fib(n): # fonction interne recursive sans appels redondants
        if n not in memo:
            if n == 0:
                memo[0] = 0
            elif n == 1:
                memo[1] = 1
            else:
                memo[n] = fib(n-1) + fib(n-2)
        return memo[n]
    return fib(n)
```

Cours 3

Chapitre 2 - Programmation des formules récursives : Redondance, Mémoïsation et localité

- ***Programmation efficace de schémas récursifs***
  - *Technique de **memoisation** pour éliminer redondance*
    - **Programmation foctionnelle (map, reduce, memoize**)
  - Technique de **blocking** pour la localité (cache)

- **Introduction à la programmation dynamique**
  - Optimisation discrète (exemple)
  - Caractérisation récursive d'une solution optimale

Automatisation/Généralisation: Un peu de fonctionnel

- def map(f, liste):
  return [f(x) for x in liste]

- def square(x) :
  return x*x
  print map(square, [0,1,2,3])
  [0, 1, 4, 9]

- map(lambda x: x*x , [0,1,2,3])

Map et ordre supérieur

- def my_map(f):
    def map_f (liste):
        return [f(x) for x in liste]
    return map_f

- mapsquare = my_map(square)
- Print mapsquare([0,1,2,3])

Exercice: reduce, filter

Memoïsation: ordre supérieur

- ```
  def memoize( f ) :
      memo = {}
      def memo_appel(x):
          if x not in memo:
              memo[x] = f(x)
          return memo[x]
      return memo_appel  # retourne la fonction!
  ```
- ~~fibKO = memoize(fib)~~ **fib = memoize(fib)**
- ~~fibKO(40)~~ fib(40)
  ```

Décorateur **@** en Python

- **@**memoize
  ```
  def fib(n):
    if n == 0:
      return 0
    elif n == 1:
      return 1
    else: return fib(n-1) + fib(n-2)
  ```

Ce qu’on a vu

- Coût = travail + défauts de cache (TP)!
- Certains problèmes d’optimisation discrète se caractérisent naturellement de manière récursive
- Elimination de redondance dans les programmes récursifs
  - Par ordonnancement des calculs
  - Par tabulation (mémoïsation)

Exemple: Fibonacci - Bilan

- Algorithme récursif initial effectue > $2^{n/2}$ ops
  - Impossible de calculer fi(200) …
  - L’univers observable aurait-il pu calculer ainsi fib(1000)?

- La mémoïsation réduit le nombre à O(n) !!!
  - Espace mémoire requis : n

- Analyse des dépendances d’appel:
  - 2 cases mémoire suffisent :
    - on retrouve l’algorithme itératif classique!

- *Question subsidiaire: comment calculer fib(n) en O(*log *n) ops sur des entiers ? (une piste: dépendances algébriques…)*

Cours 3

Chapitre 2 - Programmation des formules récursives : Redondance, Mémoïsation et localité

- ***Programmation efficace de schémas récursifs***
  - *- Technique de* ***memoisation*** *pour éliminer redondance*
    - *- Programmation foctionnelle (map, reduce, memoize)*
  - - **Technique de blocking pour la localité (cache)**

- **Introduction à la programmation dynamique**
  - - Optimisation discrète (exemple)
  - - Caractérisation récursive d'une solution optimale

Mémoisation…
et blocking pour la localité!

- Fibo avec tableau de memoisation
- Coefficients binomiaux

Fibo avec mémoïsation : analyse défauts de cache

```
def fib_memo_basique(n):
    memo = {} #table de memorisation du résultat des appels
    def fib(n): # fonction interne recursive sans appels redondants
        if n not in memo:
            if n == 0:
                memo[0] = 0
            elif n == 1:
                memo[1] = 1
            else:
                memo[n] = fib(n-1) + fib(n-2)
        return memo[n]
    return fib(n)
```

Analyse défauts:

- Un seul parcours du tableau memo de 0 à n
- #defauts = n/L

Exemple: coefficients du binôme

$$\begin{cases} C_n^p = C_{n-1}^p + C_{n-1}^{p-1} & 0 < p < n \\ C_n^0 = C_n^n = 1 \end{cases}$$

- Arrangements de p valeurs parmi n

- Exercice: écrire un programme qui calcule C(n,p) avec mémoïsation.
  - Donner le coût

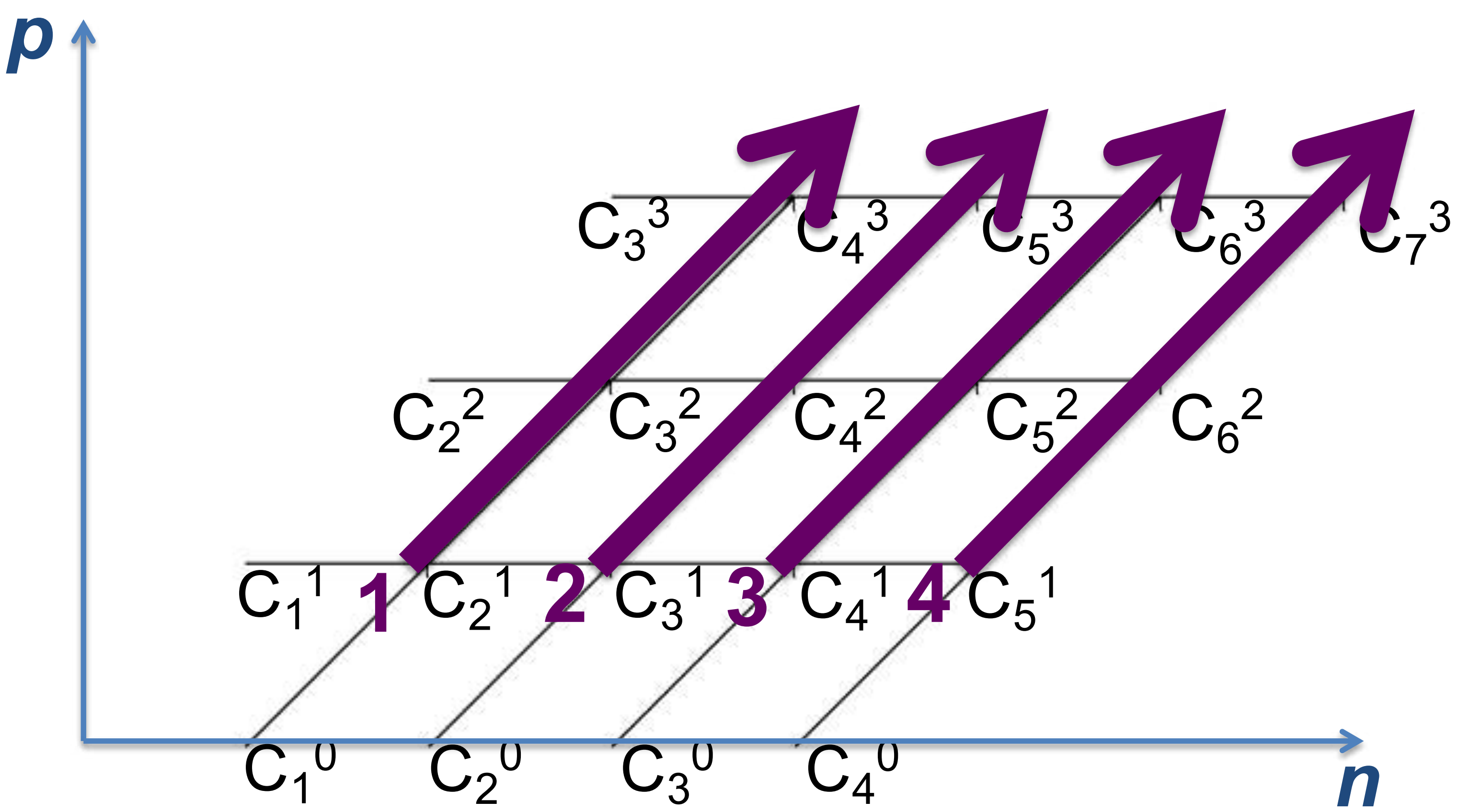
Coefficients binomiaux
C(n,p)

C_n^p Cache aware (1)

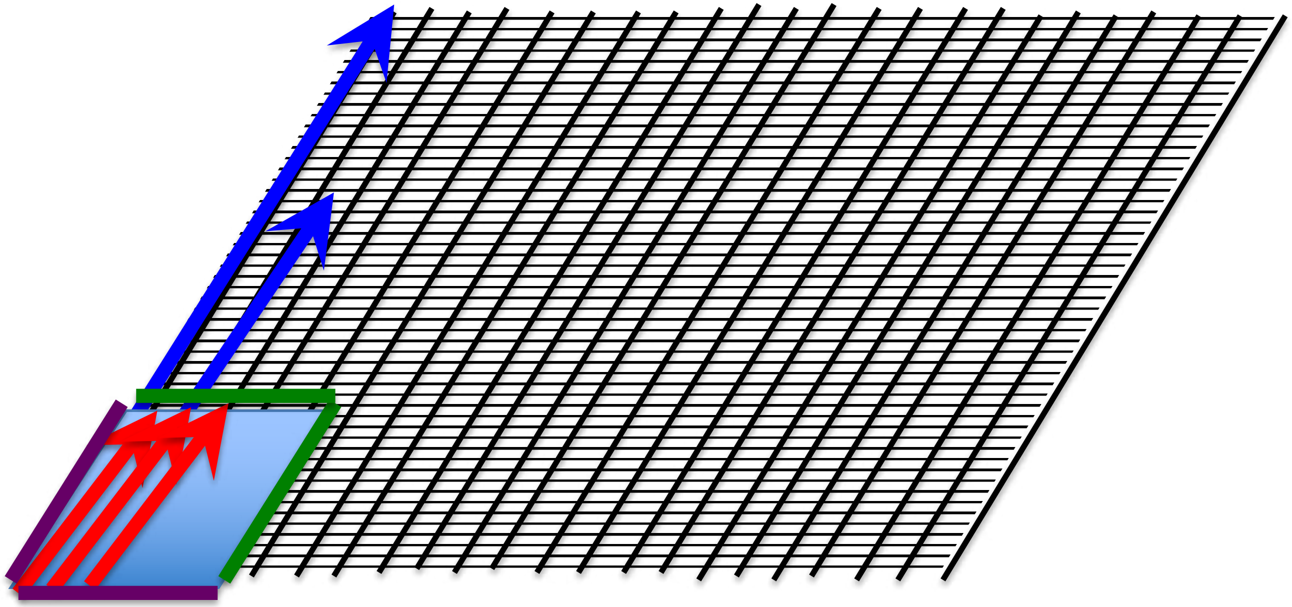
- Graphe de dépendance en stencils (en plusieurs dimensions): schémas de calcul fréquents :
  - Cnp (déjà vu)
  - Calcul numérique (Jacobi)
  - Exemples similaires en programmation dynamique
    - Patch optimal, distance de Fréchet, …

- **Ici** : calcul du Cnp par calcul itératif par diagonale en supposant $p < n-p$ (sinon par colonne):
  - Si Z assez grand : Q(n,p,L,Z)= p/L défauts de cache ☺
    - Une fois la diagonale de taille p en cache, plus de défauts!
  - Si Z petit (Z<p): Q(n, p, L, Z)= (n-p).p/L défauts cache☹
    - En bas de la hiérarchie (niveau L1), Z est petit…

C_n^p Cache aware (2.a)

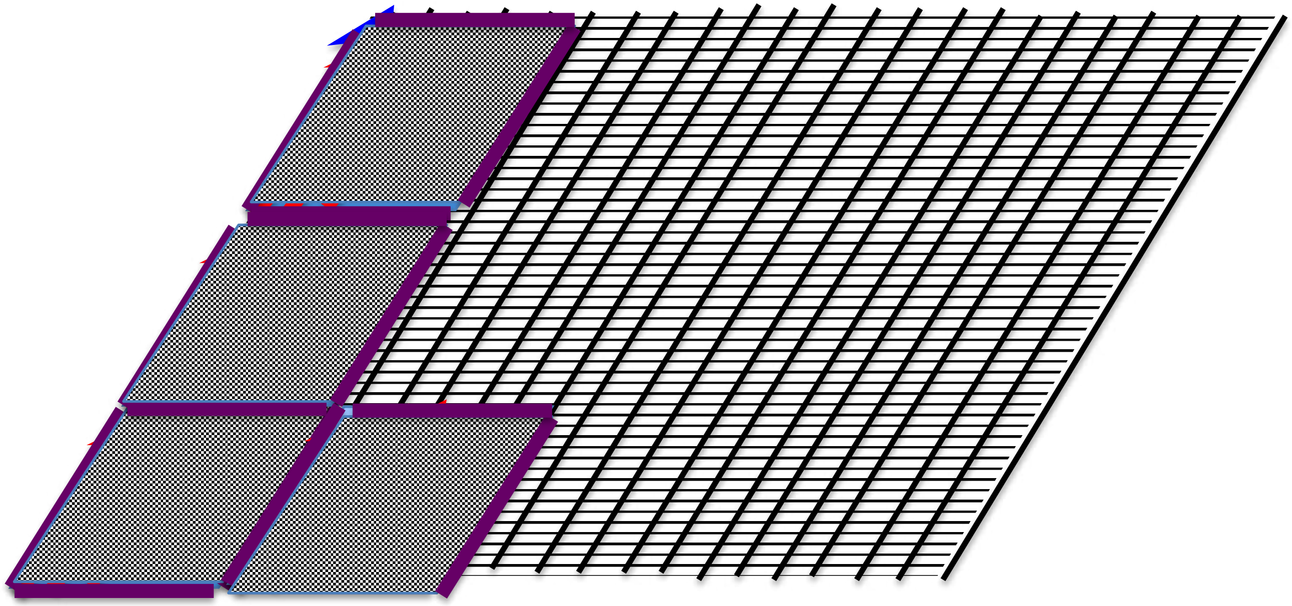


C_n^p Cache aware (2.b)



- En faisant les calculs par blocs tenant dans le cache :
  - Il suffit de stocker les frontières du blo:c **read E-S** ; **write W-N**
  - Pour minimiser le nombre de points sur les frontières: prendre des bloc carrés K x K qui s'exécutent dans le cache de taille Z
    - Calcul en place avec les bords S et E => $2.K \leq Z+O(1)$ convient
    - Seulement $2K/L + O(1)$ cache miss pour K^2 calculs
    - $(n-p).p/K^2$ blocs **=>** $4(n-p).p/K.(1/L+1/K)=$**$4(.n-p).p/(Z.L)$** défauts

C_n^p Cache aware (2.c)



- Avec K = Z/2 :   Q(n, p, L, Z) = O ((n-p).p / (L.Z)) ☺
- Le calcul par blocs permet le parallélisme sur la boucle externe! ☺

Cnp cache oblivious

- découpe récursive par blocs
  - On ne fait des calculs qu'au seuil d'arrêt !
    Le reste n'est que des appels récursifs sans accès aux tableaux (juste découpe=arithmétique de pointeurs)
  - On s'arrête a un seuil assez petit; on peut unroller la boucle sur le bloc qui concentre tout le coût du calcul et les accès mémoire !
- La découpe récursive (en 2 selon la plus grosse dimension) tire parti de la hiérarchie mémoire
  - Au bout d'un moment on fini par tomber dans le cache visé, à tout niveau de la hiérarchie!
  - Q= O((n-p).p / (L.Z)) sans connaître L et Z !

Cours 3

Chapitre 2 - Programmation des formules récursives : Redondance, Mémoïsation et localité

- ***Programmation efficace de schémas récursifs***
  - *- Technique de **memoisation** pour éliminer redondance*
    - *- Programmation foctionnelle (map, reduce, memoize)*
  - *- Technique de **blocking** pour la localité (cache)*

- **Introduction à la programmation dynamique**
  - Optimisation discrète (exemple)
  - Caractérisation récursive d'une solution optimale

Introduction à la programmation dynamique

Caractérisation récursive de l'optimum

i.e. la valeur de(s) solution(s) optimales(s)

Exemple de problème d’optimisation

- n éléments et une relation de compatibilité
  - Ex. Loup, Salade, Chèvre, Hirondelle
- Chaque élément apporte un gain

- Question: Trouver un sous-ensemble de p éléments et apportant un gain maximal.
  - a/ Si tous les éléments sont compatibles
  - b/ Le booléen R(i,j) indique si i est compatible avec j

Problème du sac à dos binaire

- Soit un sac de volume V, et n articles différents
  - Soit u_k l'utiité (bénéfice) du $k^{ième}$ article, et v_k son volume
- Quelle est l'utilité maximale?

Variante: sac à dos général: on dispose de chaque objet à volonté

Problème du sac à dos binaire

- Récursion, donc indices à récurer…
  - n = nombre d'objets restants à examiner ($1 \leq n \leq n_{init}$)
  - V = volume libre dans le sac ($0 \leq V \leq V_{init}$)
  - Déf: $U_{V,n}$ = utilité maximale d'un sac de volume V qu'on peut remplir avec les objets 1 à n
  - Formule récursive:

$$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n,n-1}, \hat{U}_{V,n-1}) & \textbf{si } v_n \leq V \\ \hat{U}_{V,n-1} & \textbf{sinon} \end{cases}$$

$$\hat{U}_{V,0} = 0$$

Compilation naïve

$$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n,n-1}, \hat{U}_{V,n-1}) & \text{si } v_n \leq V \\ \hat{U}_{V,n-1} & \text{sinon} \end{cases}$$

$$\hat{U}_{V,0} = 0$$

```
double sacMax( int n, int V) {
   |
   |  if (n==0) { res = 0; }
   |  else if (v[n] ≤ V) {
   |    max1 = u[n] + sacMax( V – v[n], n-1) ;
   |    max2 = sacMax( V, n-1 ) ;
   |    if (max1 > max2) { res = max1 ; }
   |    else             { res = max2 ; }
   |  }
   |  else { res= sacMax( V, n-1 ) ; }
   |
   |  return res;
}
```

Mémoïsation

$$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n,n-1}, \hat{U}_{V,n-1}) & \text{si } v_n \leq V \\ \hat{U}_{V,n-1} & \text{sinon} \end{cases}$$

$$\hat{U}_{V,0} = 0$$

```
double sacMaxMemoisation( int n, int V) {
    double memoI[…,…] ; // déclaration tableau temporaire
      double sacMax( int n, int V) {
        | if (memo [n, V] ≠ -1) return memo[n, V] ;
        | if (n==0) { res = 0; }
        | else if (v[n] ≤ V) {
        |   max1 = u[n] + sacMax( V – v[n], n-1) ;
        |   max2 = sacMax( V, n-1 ) ;
        |   if (max1 > max2) { res = max1 ; }
        |   else             { res = max2 ; }
        | }
        | else { res= sacMax( V, n-1 ) ; }
        | memo[n, V] = res ;
        | return res;
      }
  // main
  memo = allocate_and_init(n , V , -1) ;
  return sacMax(n, V) ;
}
```

Analyse de coût

- Version naïve : en pire cas, objets de volume 1
  $T(n,V)=T(n-1,V-1) + T(n-1,V) + O(1)$
  … exponentiel [≥Fibo(m) en posant m=n+V …]

- Version avec mémoïsation (cas V > n) :
  il y a moins de n.V appels, chacun de coût constant
  Donc $T(n,V) \leq O(n . V)$

- Prise en compte des défauts de cache:
  réordonnancement des calculs en « blocs »
  - Itération par blocs (cache aware)
  - Découpe récursive en blocs plus petits (cache oblivious)

Cageots de fraises

- **n** cageots de fraises doivent être distribués dans **k** magasins
- Les bénéfices que l'on peut retirer de chaque magasin dépendent du nombre de cageot de fraises sont fournis
  - G(i,j) : gain espéré si l'on met i cageots en vente dans le magasin j
- Quel est le gain maximal espéré
  - Caractérisation récursive du gain espéré optimal
  - Ecrire un programme avec mémoïsation

- *Prochain cours : combien de cageots mettre dans chaque magasin?*

Fin COURS 3 : ce qui a été vu…

- Programmes récursifs
  - Elimination de calculs redondants par tabulation (***mémoïsation***)
- Amélioration de la localité par ***blocking***
  - D'abord bloc-itératif (cache aware)
  - Puis bloc-récursif (cache oblivious)
- Caractérisation récursive de la valeur optimale de certains problèmes d'optimisation discrète
  - Équation de Bellman -> **programmation dynamique**