# Qualité algorithmique (R5.A.04)
# Partie II : Programmation dynamique

Marin Bougeret
LIRMM, IUT/Université de Montpellier

# Outline

# Histoire et Contexte

- la programmation dynamique est une technique générale permettant de réduire drastiquement (typiquement de exponentiel à polynomial) la complexité d'un algorithme récursif.
- c'est une technique extrêmement répandue (algos poly, FPT, d'approximation..) et très puissante !
- elle permet de calculer des fonctions, résoudre des problèmes d'optimisation, de décision..

- Dans les années 50 Richard Bellman choisit le terme programmation dynamique pour plaire à son supérieur
- .. mais cela n'a pas grand chose à voir avec le dynamisme ;)

# Exemple 1 : Fibonacci

Définition de la suite de Fibonacci :

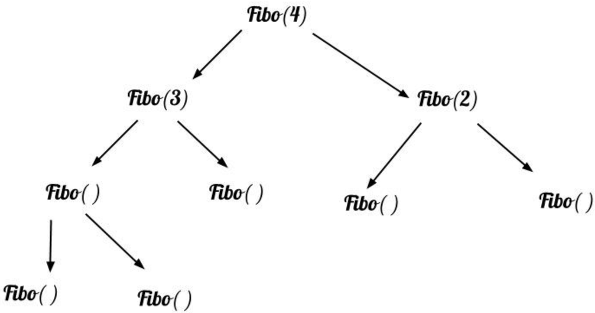
$$F(0) = 1;\ F(1) = 1; \quad F(n) = F(n-1) + F(n-2)$$

```
int Fib(n){
  if (n<=1) return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

Complexité : $Fib(n)$ est exponentielle en $n$.
Cause : Multiplicité de calcul d'un même nombre

# Où est le problème ?

La figure représente le calcule de l'algorithme récursif de Fib(4).
Completez-la et marquez les calculs faits en double



# Clé : mémoïsation

## solution

- pour éviter de refaire des calculs déjà faits, mémoriser les résultats à l'aide d'un tableau.
- étant donné $n$, on déclare un tableau $t$ de $n$ cases qu'on initialise à $-\infty$
- puis, on appelle FibDP (Dynamic Programming)

# Clé : mémoïsation

```
int FibClient(n){
   t = new int[n]; //var globale
   pour tout i, t[i]= -infini;
   return FibDP(t,n);
}

int FibDP(t,n){
  if(t[n]==-inf){
    int res;
    ancien code en remplacant (return truc ;)
       par (res = truc;)
    t[n]=res;
  }
  return t[n];
}
```

On peut prouver facilement que FibDP(n) = Fib(n)

# Clé : mémoïsation

```
int FibClient(n){
   t = new int[n]; //var globale
   pour tout i, t[i]= -infini;
   return FibDP(t,n);
}

int FibDP(t,n){
  if(t[n]==-inf){
    int res;
    if (n<=1) res = 1;
    else res = FibDP(t,n-1)+FibDP(t,n-2);
    t[n]=res;
  }
  return t[n];
}
```

On peut prouver facilement que FibClient(n) = Fib(n)

# Complexité d'une DP

**Théorème : temps**

La complexité d'une DP (qui termine) basée sur un tableau $t$ est

$$\mathcal{O}(taille(t) \times c)$$

avec $c$ la complexité d'un appel en comptant $\mathcal{O}(1)$ pour chaque appel recursif.

# Complexité d'une DP

**Théorème : temps**

La complexité d'une DP (qui termine) basée sur un tableau $t$ est

$$\mathcal{O}(taille(t) \times c)$$

avec $c$ la complexité d'un appel en comptant $\mathcal{O}(1)$ pour chaque appel recursif.

Autrement dit, on compte comme si l'on calculait chaque case du tableau une seule fois, et que les appels récursifs comptaient $\mathcal{O}(1)$.

# Complexité d'une DP

**Théorème : temps**

La complexité d'une DP (qui termine) basée sur un tableau $t$ est

$$\mathcal{O}(taille(t) \times c)$$

avec $c$ la complexité d'un appel en comptant $\mathcal{O}(1)$ pour chaque appel recursif.

Autrement dit, on compte comme si l'on calculait chaque case du tableau une seule fois, et que les appels récursifs comptaient $\mathcal{O}(1)$.

Contrairement au backtracking, on a une garantie théorique sur la complexité !

# Application à FibDP

```
int FibDP(t,n){
  if(t[n]==-inf){
    int res;
    if (n<=2) res = 1;
    else res = FibDP(t,n-1)+FibDP(t,n-2);
      //O(1)
    t[n]=res;
  }
  return t[n];
}
```

- Complexité de FibDP(n) : taille(t)=$\mathcal{O}(n)$, $c = 1 \Rightarrow \mathcal{O}(n)$
- Mémoire utilisée : $\mathcal{O}(n)$

**What's next**

Essayons d'appliquer cette technique à nos algorithmes de branchements pour des problèmes difficiles.

# Outline

1. Programmation Dynamique (DP)
2. Application au Sac à dos (KP : KnaPsack)
3. Application au plus court chemin

# Sac à dos (KP : KnaPsack)

## Problème Sac à dos

- entrée : un entier $C$, et deux tableaux d'entiers $p, v$ de taille $n$
  - $C$ représente la taille du sac à dos
  - l'objet $i$ occupe une place $p[i]$ dans le sac, mais rapporte une valeur $v[i]$
- sortie : un sous ensemble $S$ d'objets tenant dans le sac ($p(S) \leq C$, avec $p(S) = \sum_{i \in S} p[i]$)
- fonction objectif : maximiser $v(S)$, avec $v(S) = \sum_{i \in S} v[i]$

- Ce problème est "dans la catégorie difficile" (pas d'espoir d'algo poly en $n$)
- Brute force : couterait (au moins) $2^n$
- On va le résoudre en $\mathcal{O}(nC)$

# Sac à dos (KP : KnaPsack)

## Branchement naturel

- Modélisation d'une solution : $n$ variables booléenes $x_i$ (prendre objet $i$ ou pas).
- Branchement : pour chaque objet, essayer de le prendre ou pas.

# Sac à dos (KP : KnaPsack)

## Branchement naturel

- Modélisation d'une solution : $n$ variables booléenes $x_i$ (prendre objet $i$ ou pas).
- Branchement : pour chaque objet, essayer de le prendre ou pas.

## Réduire les paramètres

- Ce branchement est exactement comme ceux que l'on faisait en backtracking. modélisation d'une solution : $n$ variables booléenes $x_i$ (prendre objet $i$ ou pas)
- La différence est dans ce que l'on donne en paramètre de l'algorithme :
  - si l'on donnait $(s, D)$ (solution partielle, domaine) ou même uniquement la liste des objets déjà pris
  - alors la taille du tableau de mémoïsation serait ENORME (au moins $2^n$)
  - point clef d'une DP efficace : il faut donner "le minimum d'information utile aux appels récursifs"

# Sac à dos (KP : KnaPsack)

## Branchement naturel

- Modélisation d'une solution : $n$ variables booléenes $x_i$ (prendre objet $i$ ou pas).
- Branchement : pour chaque objet, essayer de le prendre ou pas.

## Réduire les paramètres

- Ici, quand on vient de brancher et de choisir l'objet 0..
- Ce qui importe pour la récurrence, c'est juste de savoir :
  - qu'elle doit traiter les objets $i \geq 1$
  - qu'il lui reste une place $c = C - p[0]$

# Sac à dos (KP : KnaPsack)

D'où la spécification (pour l'instant on calcule juste la valeur max)

```
int sacAux(int[]p, int[]v, int c,int i){
// prerequis
  // 0 <= c <= C
  // 0 <= i <= n (n: nb d'objets)
// action
  // calcule la valeur maximale qu'on peut
     mettre dans un sac de taille c avec les
     objets >= i
```

Et le problème associé (que sacAux résoud)

**Problème Sac à dos Aux**

- entrée : $(p, v, C)$ comme dans Sac à dos, et $c$, $i$, avec $0 \leq c \leq C$ et $0 \leq i \leq n$
- sortie : un sous ensemble $S$ d'objets $\geq i$ tenant dans un sac de taille $c$
- fonction objectif : maximiser $v(S)$

```
int sacAux(int[]p, int[]v, int c,int i){
  if(i==n){ return 0}
  else
    if(p[i] > c){
      return sacAux(c,i+1)
    }
    else{
      return max(v[i]+sacAux(p,v,c-p[i],i+1),
                 sacAux(p,v,c,i+1))
    }
}
```

Allez, une dernière fois : DP associée

```
int sacAuxDP(t, int[]p, int[]v, int c,int i){
  if(t[c,i]==-inf){
    int res;
    if(i==n){ res= 0}
    else{
      if(p[i] > c){
        res = sacAuxDP(c,i+1)
      }
      else{
        res = max(v[i]+sacAuxDP(t,p,v,c-p[i],i
            +1),sacAuxDP(t,p,v,c,i+1))
      }
    }
    t[c,i] = res;
  }
  return t[c,i];
}
```

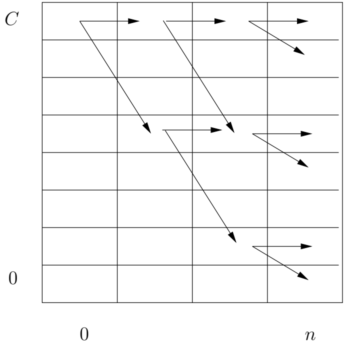
Allez, une dernière fois : DP associée

```
int sacClient(int[]p, int[]v,int C){
  n = p.length;
  int[][]t = new int[C+1][n+1];
  //init t à -inf
  return sacAuxDP(t,p,c,C,0);
}
```

# Complexité

D'après le Théorème, sacAuxDP s'execute en temps $(C+1)(n+1) \times \mathcal{O}(1) = \mathcal{O}(Cn)$.

Que se passe t-il vraiment quand on lance sacAuxDP(C,0) avec $C = 7,\ n = 3,\ p[0] = 3, p[1] = 3, p[2] = 1$ ?



# Complexité

D'après le Théorème, sacAuxDP s'execute en temps $(C+1)(n+1) \times \mathcal{O}(1) = \mathcal{O}(Cn)$.

Que se passe t-il vraiment quand on lance sacAuxDP(C,0) avec $C = 7,\ n = 3,\ p[0] = 3, p[1] = 3, p[2] = 1$ ?

- $\mathcal{O}(Cn)$ est donc un pire cas (comme si on avait calculé une fois toutes les cases), en pratique on en fait moins (mais on a du mal à dire combien exactement)
- si l'on devait éxecuter à la main .. on utiliserait aussi un tel tableau pour éviter de recalculer les appels !

# Ex 2 : Sac à dos (KP : KnaPsack)

**Remarques**

- On a l'impression de faire un brute force .. pourquoi intuitivement a-t-on une complexité faible ?
- Car on a réussi à encoder efficacement la situation restante après chaque choix.

Ex de mauvais encodage : se rappeler de tous les objets j<i que l'on a déjà pris

```
int sac(list l,int i){
  //prerequis :
    // 0 <= i <= n (n: nb d'objets)
    // l ne contient que des j < i
    // le sac contient deja tous les objets de
       l
  //action :
    // calcule la valeur maximale qu'on peut
       mettre dans la place restante avec les
       objets >= i
```

### Remarques

Intuitivement : le (bon) encodage est efficace car de nombreuses "trajectoires" aboutissent à la même situation. Par exemple, on peut avoir :

```
C->C-p[0]->C-p[0]-p[2]->C-p[0]-p[2]-p[5]  =  c
C->  ..   ->  ..     ->C-p[0]-p[2]-p[3]-p[4]  =  c
C->  ..   ->  ..          ->C-p[0]-p[1]-p[4]  =  c
   ...
C->  ..   ->  ..          ->C-p[3]-p[4]-p[5]  =  c
```

# Rmq 1 : obtenir aussi une solution

**Comment obtenir une solution (and not only la valeur optimale) ?**

Réecrire l'algorithme pour obtenir une solution est en général trivial.

Exemple

```
list sacV2(int[]p, int[]v, int c,int i){
  if(i==n){ return []}

  if(p[i] > c){
   return sacV2(c,i+1)
  }
  else{
    l1 = {i} U sacV2(p,v,c-p[i],i+1);
    l2 = sacV2(p,vc,i+1);
    if(v(l1) > v(l2))
      return l1;
    else
      return l2;
  }
}
```

# Rmq 2 : dérecursification

- On peut déduire d'une DP un algorithme itératif qui remplit le tableau "dans le bon ordre".
- On peut ensuite généralement réduire la mémoire utilisée en n'utilisant qu'un tableau plus petit.

# Rmq 2 : dérecursification

Par exemple pour sac : (penser au tableau qui se remplit colonne par colonne en partant de la droite plutôt que de lire ce code !)

```
void sacIteV1(C,p,v){
  declarer t de taille nC
  for(i from n to 0)
    for(c from 0 to n)
      if(i==n){ t[c,i]=0}
      else{
        if(p[i] > c){
          t[c,i]=t[c,i+1]
        }
        else{
          t[c,i]=max(v[i]+t[c-p[i],i+1],t[c,i
              +1]);
        }
      }
}
```

Complexité : $\mathcal{O}(nC)$

Mémoire utilisée : $\mathcal{O}(nC)$

V2 : 2 tableaux de taille $C$ seulement : on a besoin que de la colonne courante et de celle juste à sa droite.

```
void sacIteV2(C,p,v){
  declarer tprec et tcour de taille C
  init tprec avec 0
  for(i from n-1 to 0)
    for(c from 0 to n)
      if(p[i] > c){
          tcour[c,i]=tprec[c,i+1]
        }
        else{
          tcour[c,i]=max(v[i]+tprec[c-p[i],i
              +1],tprec[c,i+1]);
        }
      }
      tprec <- tcour
}
```

Complexité : $\mathcal{O}(nC)$

Mémoire utilisée : $\mathcal{O}(C+n)$

# DP Récursive Vs ou DP Iterative ?

## DP Itérative

- Avantage : on gagne en mémoire
- Inconvénients : on est sûr de remplir l'équivalent de tout le tableau, et il faut réfléchir pour trouver le bon ordre de remplissage

## DP Récursive

- Avantage : il faut juste écrire l'algo récursif (l'ajout du tableau est trivial), on ne calculera que les cases dont on aura besoin (mais il faut tout de même initialiser le tableau ! )
- Inconvénients : place mémoire

# Bilan

## Comparaison backtrack vs DP

- En Backtracking on est malin sur la propagation des contraintes (et sur l'ordre d'exploration)
- En DP on est malin sur les paramètres que l'on donne à la récursion et l'on a une garantie théorique de complexité.

Pour écrire une DP :

- on écrit un algorithme récursif (demande de réflechir ..)
- on ajoute le tableau de mémoïsation et la version cliente (ne demande pas de réflechir)

Ici on ne s'intéressera pas aux détails d'implémenation d'une DP (par ex : au lieu de prendre un tableau pour $t$, on peut prendre n'importe quelle structure de donnée (liste, hashtable, ..)).

# Outline

1. Programmation Dynamique (DP)
2. Application au Sac à dos (KP : KnaPsack)
3. Application au plus court chemin

# Exemple 2 : Déterminer le plus court chemin dans un graphe

- Nous allons étudier l'algorithme de Bellman-Ford pour le plus court chemin.
- Vous connaissez déjà l'algorithme de Dijsktra (qui est polynomial)
- La DP va nous donner un autre algorithme polynomial (encore une fois en donnant l'impressoin de juste "brancher bêtement")

# Shortest Path

Soit $G = (V, A)$ un graphe, où $V$ est l'ensemble des sommets et $A$ l'ensemble des arcs.

- Le poids de l'arc $a$ est un entier naturel $l(a)$.
- La longueur d'un chemin $P$ est égale à la somme des longueurs des arcs qui les composent (notée $l(P)$)

**Problème SP (shortest-path)**

- entrée : un graphe $G$, et deux sommet $s$ et $t$
- sortie : un $s$-$t$ chemin $P$
- objectif : minimiser $l(P)$

Remarque : on suppose qu'il n'y a pas de cycles négatifs (sinon problème mal défini .. pourquoi ?)

# Shortest Path

## Branchement

- Quel prochain sommet choisir ?
- Essayons les tous :)

# Shortest Path

```
int A(G,s,t){
//calcule longueur plus court s-t chemin
  if(s==t) return 0;
  else{
    let S(s) = ensemble des successeurs s
    return min_{v in S(s)}(l(sv)+A(G,v,t))
  }
}
```

Ecriture avec "min" raccourci d'une boucle for

```
best = infini;
for (v in S(s)){
  tmp = l(sv)+A(G,v,t);
  if(tmb < best)
    best = tmp;
}
```

# Shortest Path

```
int SP(G,s,t){
//calcule longueur plus court s-t chemin
  if(s==t) return 0;
  else{
    let S(s) = ensemble des successeurs s
    return min_{v in S(s)}(l(sv)+SP(G,v,t))
  }
}
```

Quel est le problème de A ?

- boucle infinie potentielle (à cause des cycles) !
- il n'y a rien qui devient "plus petit" lors des appels récursifs

Solution ?

- on s'impose un nombre maximum $k$ d'arêtes autorisées ..
- .. et c'est ce paramètre qui va diminuer dans les appels récursifs

## Problème SP-aux (shortest-path)

- entrée : $(G, s, t)$ comme dans shortest path, et $k$ un entier $(0 \leq k < |V(G)|)$
- sortie : un $s$-$t$ chemin $P$ ayant au plus $k$ arêtes
- objectif : minimiser $l(P)$

## Remarque

Si on sait résondre SP-aux, alors il suffira de demander à le résoudre sur $(y, n-1)$ pour avoir le $s$-$y$ chemin le plus court (car un tel chemin utilise bien au plus $n-1$ arêtes).

# Shortest Path

```
void SPaux(G,s,t,k){
 if(k==0){
   if(s==t) return 0;
   else return infini;
 }
 else{ //k != 0
   if(s==t) return 0 (car pas
       cycle negatifs)
   else
    let S(s) = ensemble des
       successeurs de s
    return min_{v in S(s)}
       l(sv)+SPaux(G,v,t,k-1)
   }
}
```

Complexité : taille tab $\times$ $\mathcal{O}(n)$... quel tab utiliserait-on ?

# Shortest Path

```
void SPAuxDP(tab,G,s,t,k){
  if(tab[s,k]==infini){
    ..


        res = l(sv)+SPAuxDP(tab,G,v,t,k-1)
   }
}
```

Complexité : tab[s][k] pour $s \in V(G)$ et $0 \leq k \leq n-1$, donc taille tab $= n^2$.

La complexité de SPAuxDP est donc $\mathcal{O}(n^3)$.

Une meilleure analyse en $\mathcal{O}(nm)$ est possible, et pas difficile.

# Conclusion

## A retenir

- DP = algo recursif + tableau (même si dérecursifiable)
- Complexité = #Cases x temps d'un appel (même si impression de brute force)
- Pour quels problèmes la DP est-t-elle un bon outil ? Situations caractéristiques où la DP est un bon outil :
  - la structure dans laquelle on cherche une solution à un ordre naturel pour la parcourir (graphe d'intervalles : de gauche à droite, arbre : de la racine vers les feuilles, pb du sac à dos : la liste d'objets, ..)
  - quand de nombreuses trajectoires (une trajectoire étant une suite de décisions locales, par pour KP, traj = $d_1, .., d_i$, avec $d_i$ vrai ssi on prend l'objet $i$) différentes peuvent aboutir au même état