Le programme

1 et 2. Rappels sur les graphes
Plus courts chemins
Programmation dynamique
(2. et) 3. Problèmes de flots
4. Problèmes d'affectation et de transport
5. Compléments et révisions

Programmation dynamique

Technique algorithmique : *générale* et puissante

Programmation dynamique : (presque) de la recherche exhaustive

où on essaie toutes les possibilités d'une manière intelligente

Programmation dynamique : sous-problèmes et reutilisation

Nombres de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$F_N = \begin{cases} F_{N-1} + F_{N-2} & \text{if } N \geq 2 \\ 1 & \text{if } N = 1 \\ 0 & \text{if } N = 0 \end{cases}$$

Objectif: calculer le Nème nombre de Fibonacci

Nombres de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

$$F_N = \begin{cases} F_{N-1} + F_{N-2} & \text{if } N \geq 2 \\ 1 & \text{if } N = 1 \\ 0 & \text{if } N = 0 \end{cases}$$

fib(N)
si N<2 : f=N
sinon f=fib(N-1)+ fib(N-2)
retourner f

Objectif: calculer le Nème nombre de Fibonacci

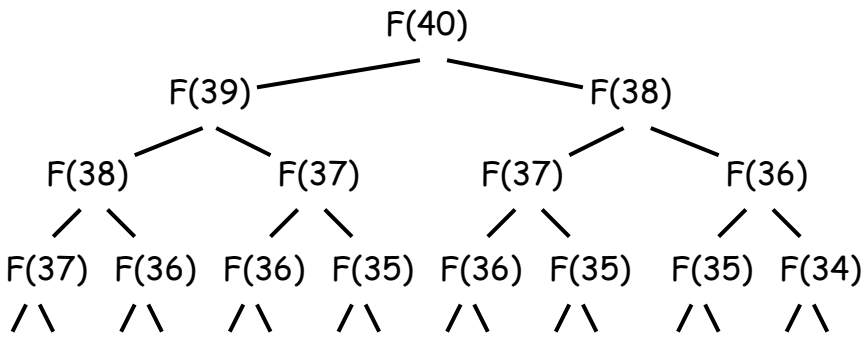
Nombres de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

$$F_N = \begin{cases} F_{N-1} + F_{N-2} & \text{if } N \geq 2 \\ 1 & \text{if } N = 1 \\ 0 & \text{if } N = 0 \end{cases}$$

fib(N)
si N<2 : f=N
sinon f=fib(N-1)+ fib(N-2)
retourner f

Objectif: calculer le Nème nombre de Fibonacci



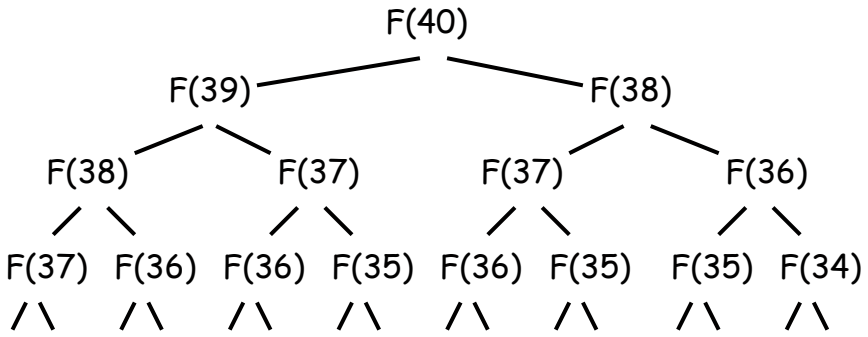
Nombres de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

$$F_N = \begin{cases} F_{N-1} + F_{N-2} & \text{if } N \geq 2 \\ 1 & \text{if } N = 1 \\ 0 & \text{if } N = 0 \end{cases}$$

fib(N)
si N<2 : f=N
sinon f=fib(N-1)+ fib(N-2)
retourner f

Objectif: calculer le Nème nombre de Fibonacci



Exponentiel !!!

Mémoisation

Algorithme de programmation dynamique avec mémoisation

```
memo={}        %un dictionnaire
fib(N):
    si N dans memo : retourner memo[N]
    si N < 2 : f = N
    sinon : f = fib(N-1)+fib(N-2)
    memo[N]=f
    retourner f
```

Mémoisation

Algorithme de programmation dynamique avec mémoisation

```
memo={}        %un dictionnaire
fib(N):
    si N dans memo : retourner memo[N]
    si N < 2 : f = N
    sinon : f = fib(N-1)+fib(N-2)
    memo[N]=f
    retourner f
```

Méthode générale pour rendre un algorithme récursif efficace

Mémoisation

Algorithme de programmation dynamique avec mémoisation

```
memo={}        %un dictionnaire
fib(N):
    si N dans memo : retourner memo[N]
    si N < 2 : f = N
    sinon : f = fib(N-1)+fib(N-2)
    memo[N]=f
    retourner f
```

Méthode générale pour rendre un algorithme récursif efficace

Pourquoi ? fib(k) fait des appels recursifs uniquement la première fois qu'elle est appelée, pour chaque k

- un appels mémoisé : temps constant
- nombre d'appels non-mémoisés : N (fib(1),…,fib(N))
- travail non-récursif à chaque appel : temps constant
- => temps en O(N)

Programmation dynamique : (presque) recursion + mémoisation

mémoiser (se rappeler) et réutiliser des solutions à des sous-problèmes qui aident à la résolution du problème initial

temps= (nombre de sous-problèmes) x (temps par sous-problème)

Pour le nombre de Fibonacci :

nombre de sous-problèmes = N
temps par sous-problème = constant (on ne compte pas les recursions)

Du bas vers le haut (Bottom-up)

Itératif(N)
f[0]=0, f[1]:=1
pour i=2 jusqu'à N: f[i]=f[i-1]+f[i-2]
retourner f[N]

Du bas vers le haut (Bottom-up)

```
Itératif(N)
f[0]=0, f[1]:=1
pour i=2 jusqu'à N: f[i]=f[i-1]+f[i-2]
retourner f[N]
```

En général,

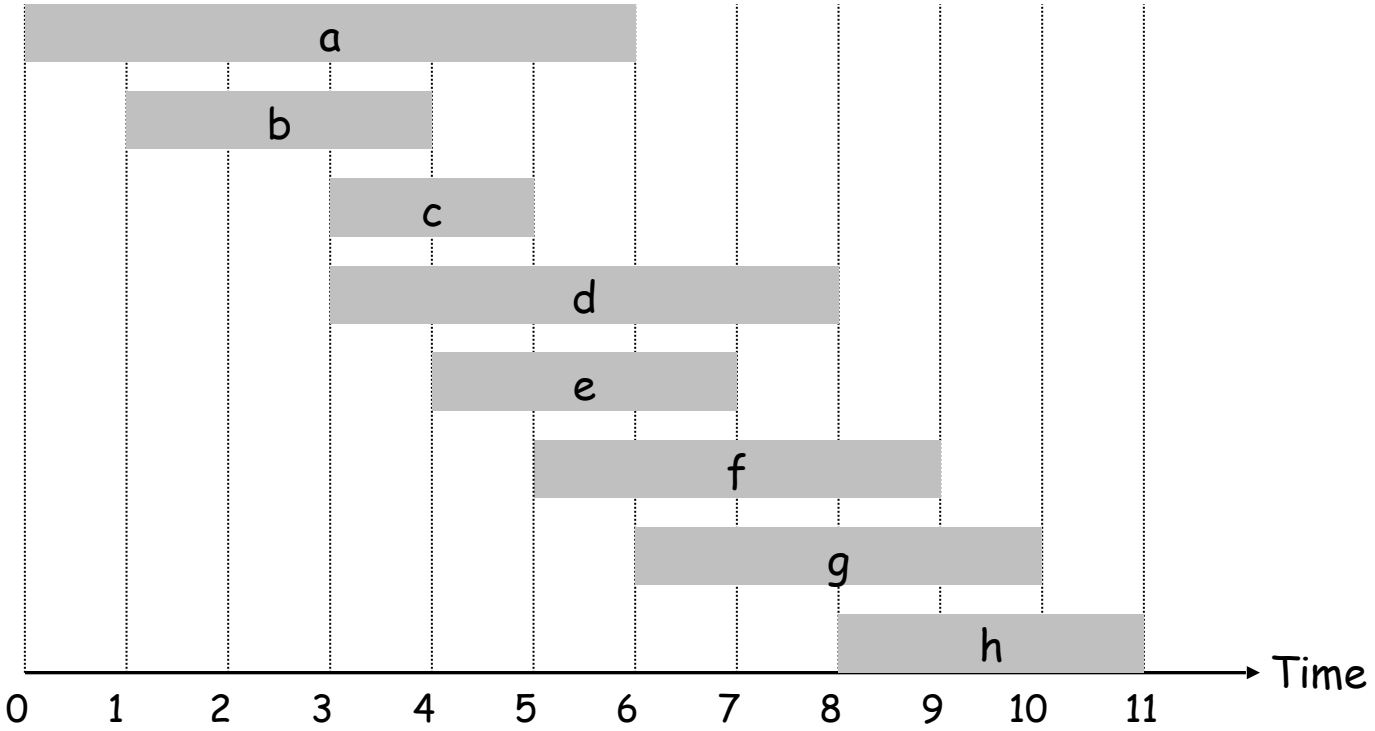
- même calcul que dans le cas de programmation dynamique mémoisée

- tri topologique du graphe orienté sans circuit des sous-problèmes

Ordonnancement d' Intervalles

Ordonnancement d'intervalles

- La tâche j commence à s_j, finit à f_j, et a un poids de valeur v_j .
- Deux tâches sont compatibles si elles ne se chevauchent pas.
- But : déterminer un sous-ensemble de tâches mutuellement compatibles de poids maximum.

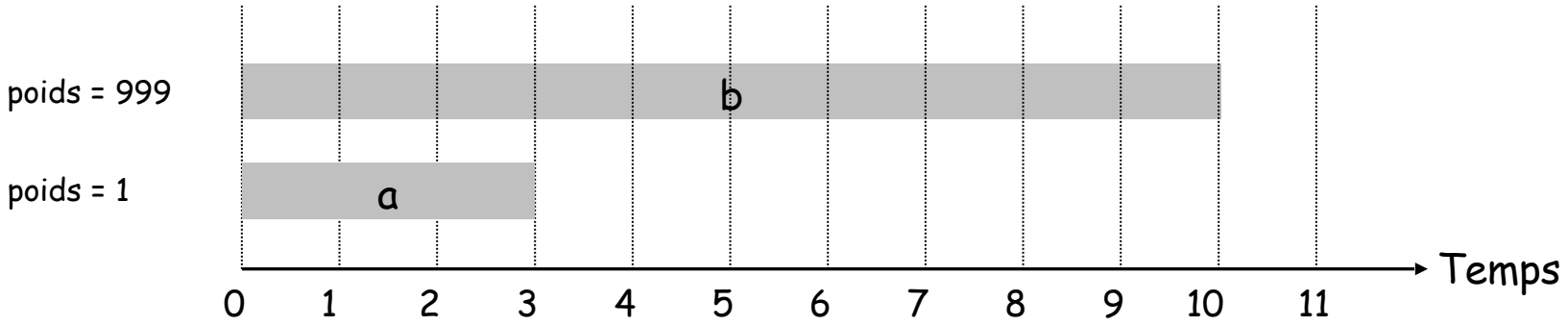


Ordonnancement d'intervalles non-pondérés

Fait. Si tous les poids sont 1, on peut utiliser glouton :

- Considérons les tâches dans l'ordre croissant de leurs dates de fin.
- Ajouter une tâche dans le sous-ensemble si elle est compatible avec les tâches déjà choisies.

Observation. Dans le cas pondéré glouton peut se tromper énormément.

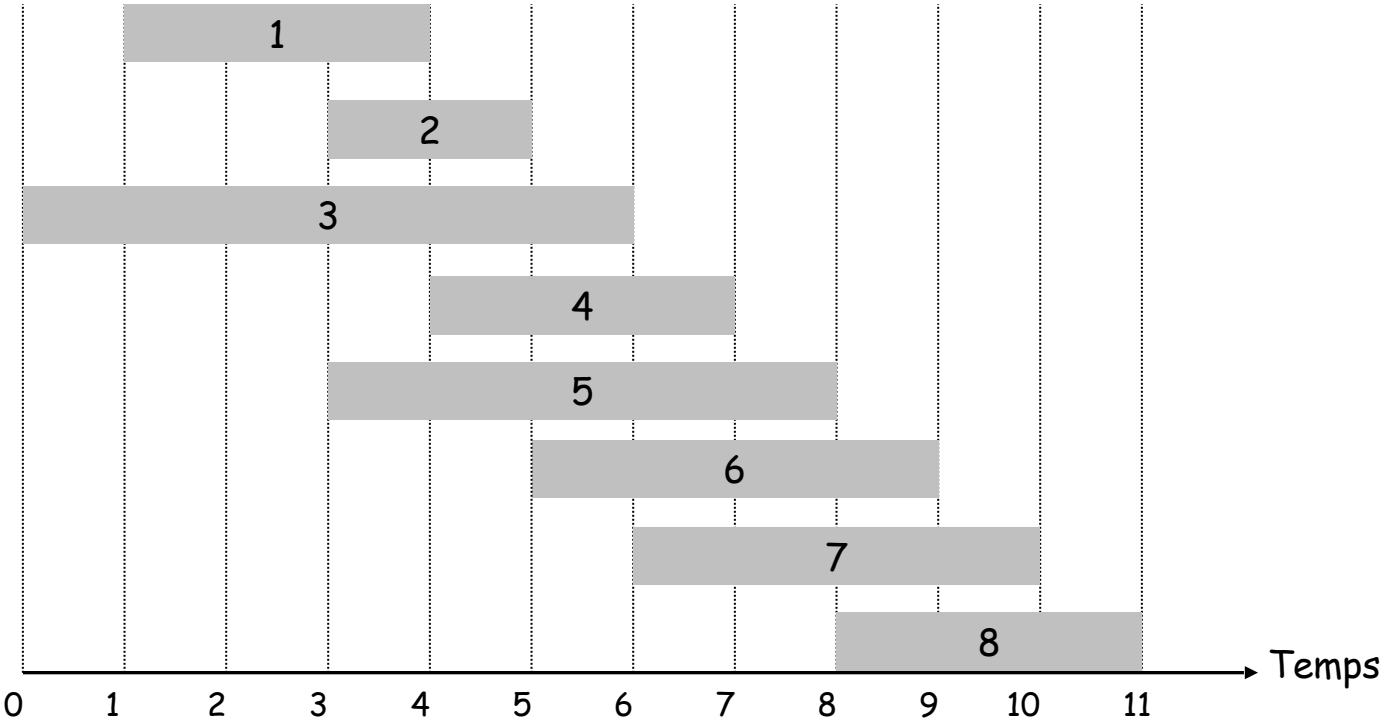


Ordonnancement d'intervalles pondérés

Notation. Numéroter selon les dates de fin : $f_1 \leq f_2 \leq \ldots \leq f_n$.

Déf. p(j) = l'indice max i < j tel que la tâche i est compatible avec j.

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



Programmation Dynamique : Choix Binaire

Notation. OPT(j) = valeur de la solution optimale pour le problème avec les tâches 1, 2, ..., j.

- Cas 1 : OPT choisit la tâche j.
  - tâches incompatibles : { p(j) + 1, p(j) + 2, ..., j - 1 }
  - doit inclure la solution optimale pour le problème 1, 2, ..., p(j)

- Cas 2: OPT ne choisit pas la tâche j.
  - doit inclure la solution optimale pour le problème 1, 2, ..., j-1

sous-structure optimale

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Ordonnancement d'intervalles pondéré : recherche exhaustive

Entrée: $n, s_1,\ldots,s_n , f_1,\ldots,f_n , v_1,\ldots,v_n$

Trier tâches par dates de fin $f_1 \leq f_2 \leq \ldots \leq f_n$.

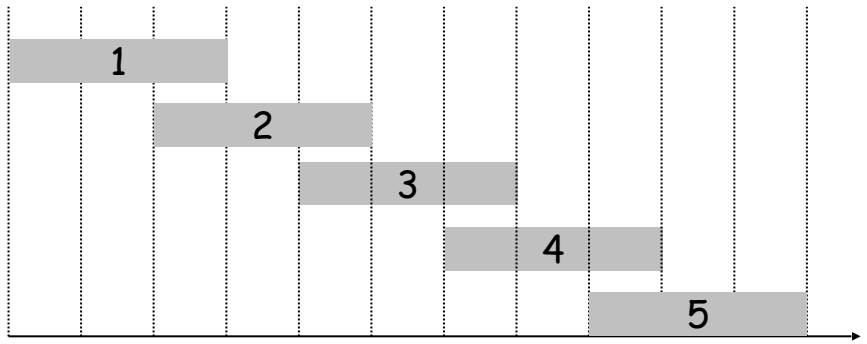
Calculer p(1), p(2), …, p(n)

```
Calculer-Opt(j) {
si (j = 0)
  retourner 0
sinon
  retourner max(v_j + Calculer-Opt(p(j)), Calculer-Opt(j-1))
}
```

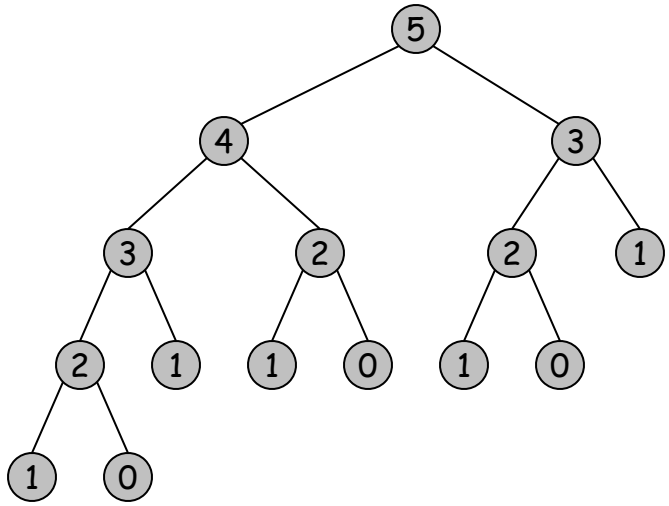
Ordonnancement d'intervalles pondérés : recherche exhaustive

Observation. algorithme exponentiel

Ex. Nombre d'appels récursifs pourrait augmenter comme dans le cas d'une suite de Fibonacci.



p(1) = 0, p(j) = j-2



Ordonnancement d'intervalles pondérés : Mémoisation

Mémoisation. Stocker les résultats de chaque sous-problème.

```
Entrée: n, s1,…,sn , f1,…,fn , v1,…,vn

Trier tâches par date de fin f1 ≤ f2 ≤ ... ≤ fn.
Calculer p(1), p(2), …, p(n)

pour j = 1 à n
   M[j] = vide      ← tableau global
M[0] = 0

M-Calculer-Opt(j) {
   si (M[j] est vide)
      M[j] = max(wj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Ordonnancement d'intervalles pondérés : Temps de calcul

Assertion. La version mémoisée prends un temps en $O(n \log n)$.

- Tri par date fin : $O(n \log n)$.
- Calculer p(·) : $O(n)$ après un tri par date de début.

- `M-Calculer-Opt(j)` : chaque appel prend un temps en $O(1)$ et soit
  - (i) retourne une valeur existante `M[j]`
  - (ii) soit il remplit une nouvelle entrée `M[j]` et fait deux appels récursifs

- Mesure de progression Φ = # d'entrée non-vides `M[]`.
  - initialement $\Phi = 0$, et $\Phi \leq n$.
  - (ii) augmente Φ de 1 $\Rightarrow$ au plus $O(n)$ appels récursifs.

- Temps total de `M-Compute-Opt(n)` est en $O(n)$. ▪

Remarque. $O(n)$ si les tâches sont triées par date de début et de fin.

Ordonnancement d'intervalles pondérés : Déterminer une solution

Q. Comment déterminer la solution ?

A. Effectuer un post-traitement

```
Faire Tourner M-Calculer-Opt(n)
Faire Tourner Déterminer-Solution(n)

Déterminer-Solution(j) {
    si (j = 0)
        retourner rien
    sinon si (v_j + M[p(j)] > M[j-1])
        print j
        Déteminer-Solution(p(j))
    sinon
        Déterminer-Solution(j-1)
}
```

- # d'appels récursifs ≤ n ⇒ $O(n)$.

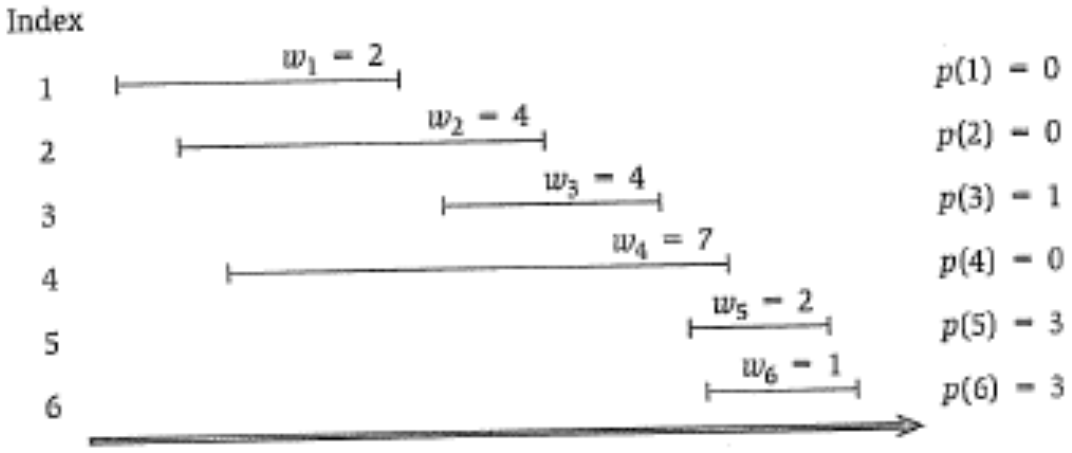
Ordonnancement d'intervalles pondérés : Bottom-Up

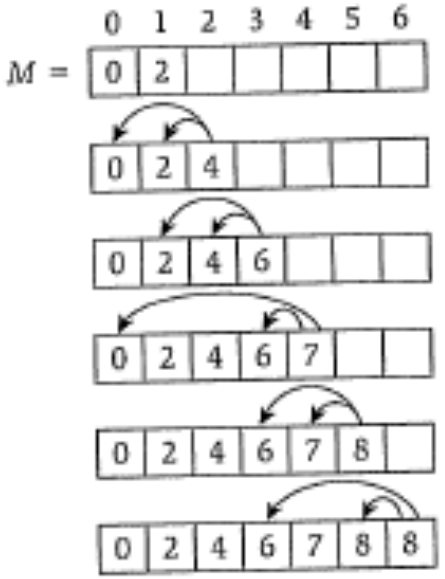
Entrée: n, $s_1,\ldots,s_n$, $f_1,\ldots,f_n$, $v_1,\ldots,v_n$

Trier les tâches par date de fin $f_1 \le f_2 \le \ldots \le f_n$.

Calculer p(1), p(2), …, p(n)

```
Itérativement-Calculer-Opt {
   M[0] = 0
   pour j = 1 à n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```





Le Problème du sac-à-dos

Le Problème du sac-à-dos

- n objets et un "sac-à-dos."
- Le poids de l'objet i est $w_i > 0$ et sa valeur est $v_i > 0$.
- Le sac-à-dos a une capacité de W.
- Objectif: remplir le sac-à-dos afin de maximiser la valeur totale

Ex: { 3, 4 } a une valeur de 40.

W = 11

objet	valeur	poids
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Glouton : ajouter à chaque fois l'objet maximisant v_i / w_i.

Ex: { 5, 2, 1 } conduit à 35 $\Rightarrow$ glouton n'est pas optimal.

Programmation Dynamique : un faux départ

Déf. OPT(i) = sous-ensemble de profit maximum de 1, …, i.

- Cas 1 : OPT ne choisit pas l'objet i.
  - OPT choisit la solution optimale de { 1, 2, …, i-1 }

- Cas 2 : OPT choisit l'objet i.
  - le choix de i n'implique pas immédiatement que nous devrons rejeter d'autres objets
  - sans connaitre quels autres objets ont été choisis avant i, nous ne connaissons pas combien de capacité reste pour i

Conclusion. nous avons besoin de plus de sous-problèmes !

Programmation Dynamique : ajout d'une nouvelle variable

Déf. OPT(i, w) = sous-ensemble de profit max profit de 1, …, i avec un poids limite de w.

- Cas 1 : OPT ne choisit pas i.
  - OPT choisit une solution optimale pour { 1, 2, …, i-1 } avec un poids limite de w

- Cas 2 : OPT choisit l'objet i.
  - nouveau poids limite = $w - w_i$
  - OPT choisit une solution optimale pour { 1, 2, …, i–1 } avec le nouveau poids limite

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i)\,\} & \text{otherwise} \end{cases}$$

Le Problème du sac-à-dos : Bottom-Up

Remplir un tableau n-par-W.

```
Entrée : n, w1,…,wN, v1,…,vN

pour w = 0 à W
   M[0, w] = 0

pour i = 1 à n
   pour w = 1 à W
      si (wi > w)
         M[i, w] = M[i-1, w]
      sinon
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

retourner M[n, W]
```

Le Problème du sac-à-dos

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
valeur = 22 + 18 = 40

W = 11

objet	valeur	poids
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Le Problème du sac-à-dos : temps d'exécution

$\Theta(n\ W)$.

- Pas polynomial par rapport à la taille de l'entrée !
- "Pseudo-polynomial."
- Version de décision du problem : problème NP-complet.

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Calculer $\lambda^*(t)$: valeur d'un PCCH de s à t

Idée :

- décomposer le problème en une séquence de sous-problèmes :
  - $\lambda(k,.)$: plus courts chemins avec au plus k arcs, k=0,…,n-1
  - Situation/état : le sommet que l'on a atteint
    - $\lambda(k,v)$: plus court chemin de s à v avec au plus k arcs.

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Calculer $\lambda^*(t)$: valeur d'un PCCH de s à t

Idée :

- décomposer le problème en une séquence de sous-problèmes :
  - $\lambda(k,.)$: plus courts chemins avec au plus k arcs, k=0,…,n-1
  - Situation/état : le sommet que l'on a atteint
    - $\lambda(k,v)$: plus court chemin de s à v avec au plus k arcs.

Algorithme :

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Calculer $\lambda^*(t)$: valeur d'un PCCH de s à t

Idée :
- décomposer le problème en une séquence de sous-problèmes :
  $\lambda(k,.)$: plus courts chemins avec au plus k arcs, k=0,…,n-1
  Situation/état : le sommet que l'on a atteint
  $\lambda(k,v)$: plus court chemin de s à v avec au plus k arcs.

Algorithme :
- On sait calculer $\lambda(0,.)$

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Calculer $\lambda^*(t)$: valeur d'un PCCH de s à t

Idée :
- décomposer le problème en une séquence de sous-problèmes :
  $\lambda(k,.)$: plus courts chemins avec au plus k arcs, k=0,…,n-1
  Situation/état : le sommet que l'on a atteint
  $\lambda(k,v)$: plus court chemin de s à v avec au plus k arcs.

Algorithme :
- On sait calculer $\lambda(0,.)$
- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Calculer $\lambda^*(t)$: valeur d'un PCCH de s à t

Idée :
- décomposer le problème en une séquence de sous-problèmes :
  $\lambda(k,.)$: plus courts chemins avec au plus k arcs, k=0,…,n-1
  Situation/état : le sommet que l'on a atteint
  $\lambda(k,v)$: plus court chemin de s à v avec au plus k arcs.

Algorithme :
- On sait calculer $\lambda(0,.)$
- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$
- $\lambda(n-1,.)$ permet de trouver la valeur cherchée : $\lambda(n-1,t)$

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Validité ?

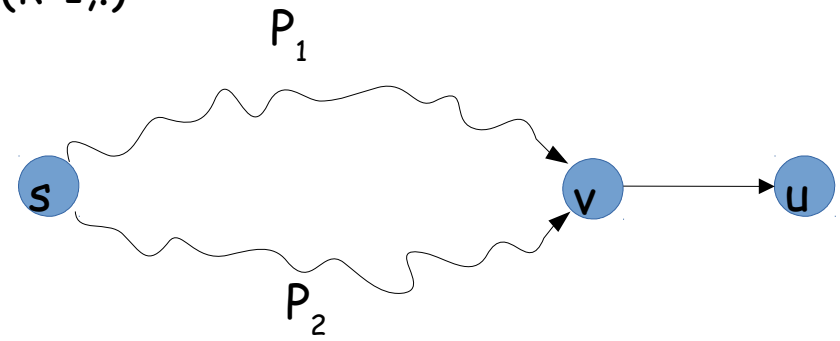
- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Validité ?

- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$

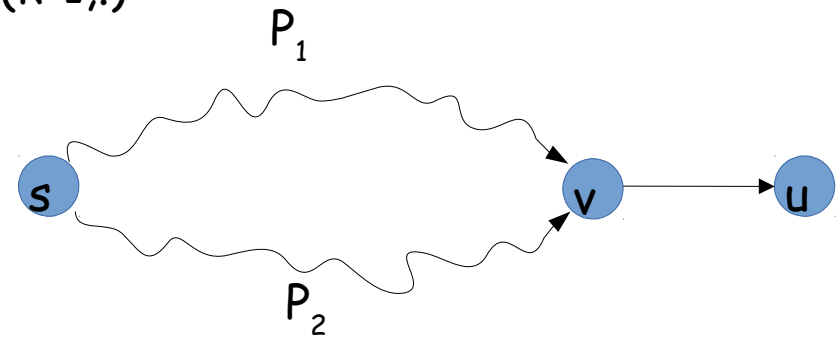


Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Validité ?

- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$



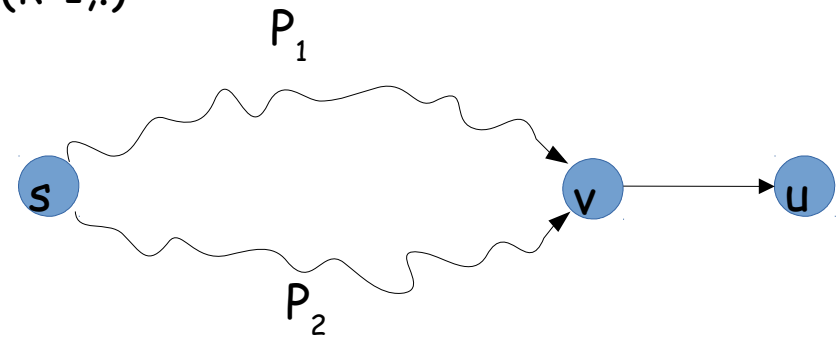
Si P est un chemin optimal de s à u, alors le chemin de s au dernier sommet visité v avant u est optimal.

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Validité ?

- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$



Si P est un chemin optimal de s à u, alors le chemin de s au dernier sommet visité v avant u est optimal.

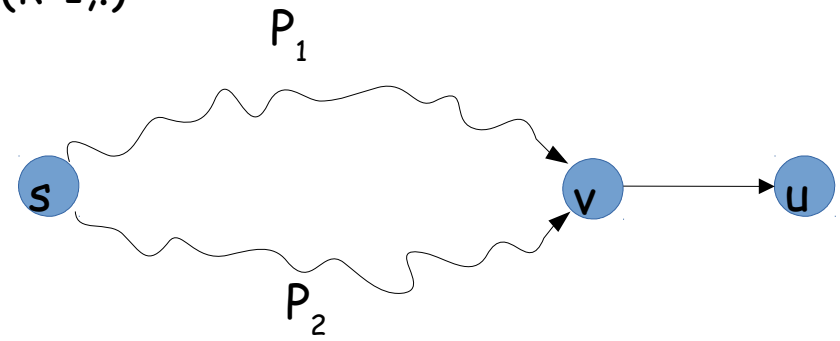
Si $d(P_1) \leq d(P_2)$, alors $d(P_1)+d(v,u) \leq d(P_2)+d(v,u)$ (un peu moins fort mais suffisant pour la validité)

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Validité ?

- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$



Si P est un chemin optimal de s à u, alors le chemin de s au dernier sommet visité v avant u est optimal.

Si $d(P_1) \leq d(P_2)$, alors $d(P_1)+d(v,u) \leq d(P_2)+d(v,u)$ (un peu moins fort mais suffisant pour la validité)

→ Il suffit de mémoriser un chemin optimal/la valeur optimale de s à v

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Complexité ?

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Complexité ?

- On décompose en n étapes $\lambda(k,.)$, $k=0,\ldots,n-1$

- A chaque étape on a n « états » (valeurs à calculer): $\lambda(k,v)$, v dans V
→ Au total n^2 valeurs à calculer

- Calcul d'une valeur : proportionnel au nombre de prédécesseur du sommet → $O(n)$.

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Complexité ?

- On décompose en n étapes $\lambda(k,.)$, $k=0,\dots,n-1$

- A chaque étape on a n « états » (valeurs à calculer): $\lambda(k,v)$, v dans V
→ Au total n^2 valeurs à calculer

- Calcul d'une valeur : proportionnel au nombre de prédécesseur du sommet → $O(n)$.

→ Complexité en $O(n^3)$.

Programmation dynamique

I. Retour sur l'algorithme de Bellman-Ford

Complexité ?

- On décompose en n étapes $\lambda(k,.)$, $k=0,\dots,n-1$

- A chaque étape on a n « états » (valeurs à calculer): $\lambda(k,v)$, v dans V
→ Au total n^2 valeurs à calculer

- Calcul d'une valeur : proportionnel au nombre de prédécesseur du sommet → $O(n)$.

→ Complexité en $O(n^3)$.

En regardant plus attentivement :à chaque étape pour l'ensemble des calculs on parcourt les arcs du graphes → complexité $O(mn)$.

II. Programmation dynamique

a) Le principe

- Décomposer le problème en une séquence de sous-problèmes indicés par k=0,…,t-1

II. Programmation dynamique

a) Le principe

- Décomposer le problème en une séquence de sous-problèmes indicés par k=0,…,t-1
- Pour chaque sous-problème :
    - un ensemble S de situations/états possibles
    - une valeur à calculer λ(k,v), v in S : meilleure manière de parvenir dans l'état v du sous-problème k.

Programmation dynamique

II. Programmation dynamique

a) Le principe

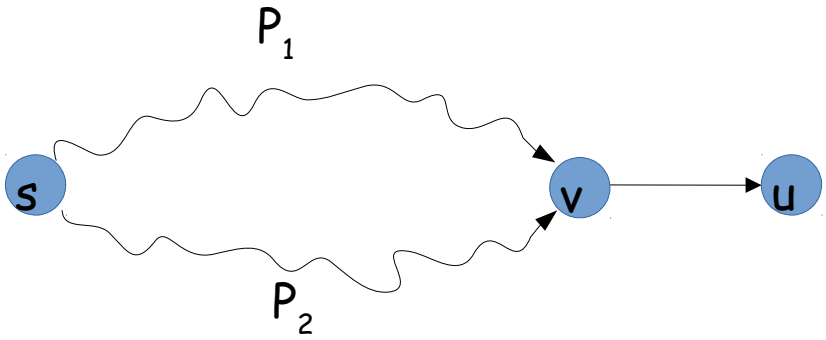
- Décomposer le problème en une séquence de sous-problèmes indicés par k=0,…,t-1
- Pour chaque sous-problème :
    - un ensemble S de situations/états possibles
    - une valeur à calculer $\lambda(k,v)$, v in S : meilleure manière de parvenir dans l'état v du sous-problème k.

Décomposition telle que :
- On sait initialiser (typiquement $\lambda(0,.)$)
- On a une relation de récurrence permettant de calculer $\lambda(k,.)$ à partir de $\lambda(k-1,.)$ (et plus généralement à partir des $\lambda(i,.), i<k$)
- Le calcul des $\lambda(k,v)$ permet de répondre au problème initial.

II. Programmation dynamique

Validité ?

Principe de Bellman : toute sous-politique d'une politique optimale est optimale



→ Permet de ne conserver que la politique/valeur optimale λ(k,v), et non pas l'ensemble des 'politiques' menant à l'état v du sous-problème k.
(→ ne pas énumérer l'ensemble des chemins, dont le nombre est exponentiel).

II. Programmation dynamique

Complexité ?

Si le calcul d'une valeur $\lambda(k,v)$ se fait en $O(p)$, alors la complexité est en $O(t|S|p)$

II. Programmation dynamique

b) Problème de chemin, revisité :
- Valeur d'un chemin = max des longueurs des arcs du chemin.
- Trouver un chemin de s à t de valeur minimale.

II. Programmation dynamique

b) Problème de chemin, revisité :
- Valeur d'un chemin = max des longueurs des arcs du chemin.
- Trouver un chemin de s à t de valeur minimale.

Algo de prog dynamique ?

II. Programmation dynamique

b) Problème de chemin, revisité :
- Valeur d'un chemin = max des longueurs des arcs du chemin.
- Trouver un chemin de s à t de valeur minimale.

Algo de prog dynamique ?

Principe de Bellman ?

Programmation dynamique

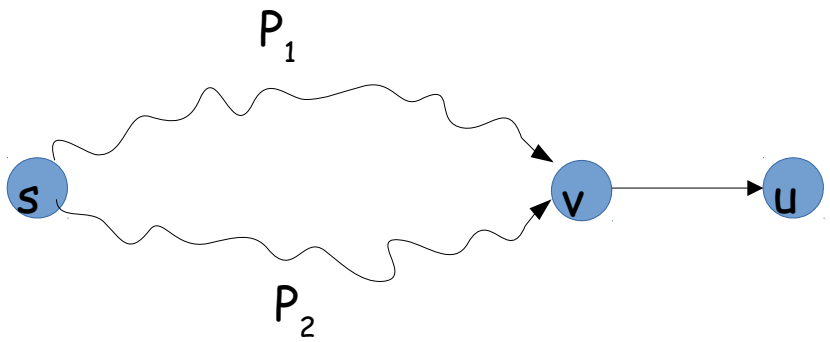
II. Programmation dynamique

b) Problème de chemin, revisité :
- Valeur d'un chemin = max des longueurs des arcs du chemin.
- Trouver un chemin de s à t de valeur minimale.

Algo de prog dynamique ?

Principe de Bellman ?

→ Principe 'relâché', 'monotonie' :
Si val(P_1) ≤ val(P_2), alors val(P_1'+'(v,u)) ≤ val(P_2'+'(v,u)) (un peu moins fort mais suffisant pour la validité)



II. Programmation dynamique

c) Exemple : gestion de stock

Une entreprise fabrique un produit à partir d'une matière première P qu'elle achète. Elle doit fabriquer chaque mois k une certaine quantité de produit nécessitant l'achat de d_k unités de P, afin de satisfaire la demande du mois.

A chaque début de mois, elle achète une certaine quantité q_k de produit P, à un prix unitaire p_k fluctuant de mois en mois.

Elle peut stocker la matière première dans un entrepôt de capacité C.

Il s'agit de déterminer une stratégie d'achat optimale de manière à satisfaire la demande et à minimiser le coût total d'achat de P.

II. Programmation dynamique

c) Exemple : gestion de stock

C=4,
Stock nul au début et à la fin

	Mois 1	Mois 2	Mois 3
Prix	3	2	4
Demande	1	3	2

II. Programmation dynamique

c) Exemple : gestion de stock

Formulation :
- Décomposition mois par mois : k=1,…, nombre de mois

C=4,
Stock nul au début et à la fin

	Mois 1	Mois 2	Mois 3
Prix	3	2	4
Demande	1	3	2

II. Programmation dynamique

c) Exemple : gestion de stock

Formulation :
- Décomposition mois par mois : k=1,…, nombre de mois
- Etat : Niveau de stock s à la fin du mois k

C=4,
Stock nul au début et à la fin

	Mois 1	Mois 2	Mois 3
Prix	3	2	4
Demande	1	3	2

II. Programmation dynamique

c) Exemple : gestion de stock

Formulation :
- Décomposition mois par mois : k=1,…, nombre de mois
- Etat : Niveau de stock s à la fin du mois k
  → $\lambda(k,s)$: politique d'achat optimale sur les k premiers mois en parvenant à un stock s à la fin du kème mois.

C=4,
Stock nul au début et à la fin

	Mois 1	Mois 2	Mois 3
Prix	3	2	4
Demande	1	3	2

II. Programmation dynamique

c) Exemple : gestion de stock

Formulation :
- Décomposition mois par mois : k=1,…, nombre de mois
- Etat : Niveau de stock s à la fin du mois k
  → $\Lambda(k,s)$: politique d'achat optimale sur les k premiers mois en parvenant à un stock s à la fin du kème mois.

Récurrence : si j'achète q_k en début de mois k :

C=4,
Stock nul au début et à la fin

	Mois 1	Mois 2	Mois 3
Prix	3	2	4
Demande	1	3	2

II. Programmation dynamique

c) Exemple : gestion de stock

Formulation :
- Décomposition mois par mois : k=1,…, nombre de mois
- Etat : Niveau de stock s à la fin du mois k
  → $\Lambda(k,s)$: politique d'achat optimale sur les k premiers mois en parvenant à un stock s à la fin du kème mois.

Récurrence : si j'achète q_k en début de mois k :
→ Coût = $p_k q_k$

C=4,
Stock nul au début et à la fin

	Mois 1	Mois 2	Mois 3
Prix	3	2	4
Demande	1	3	2

Programmation dynamique

II. Programmation dynamique

c) Exemple : gestion de stock

Formulation :
- Décomposition mois par mois : k=1,…, nombre de mois
- Etat : Niveau de stock s à la fin du mois k
  → $\Lambda(k,s)$: politique d'achat optimale sur les k premiers mois en parvenant à un stock s à la fin du kème mois.

Récurrence : si j'achète q_k en début de mois k :
→ Coût = $p_k q_k$
→ Stock à la fin du mois k: $s_k = s_{k-1} + q_k - d_k$
Contrainte : $s_{k-1} + q_k \leq C$
→ Relation de récurrence en regardant les niveaux de stocks le mois précédent.

C=4,
Stock nul au début et à la fin

	Mois 1	Mois 2	Mois 3
Prix	3	2	4
Demande	1	3	2

II. Programmation dynamique

d) Calcul

i) Bottom-up (de bas en haut) :

→ Calcul itératif :

II. Programmation dynamique

d) Calcul

i) Bottom-up (de bas en haut) :

→ Calcul itératif :

- Initialisation : cas de bases (typiquement $\lambda(0,v)$)
- Pour k de 1 à t-1 : calcul des $\lambda(k,v)$ en utilisant la relation de récurrence.

II. Programmation dynamique

d) Calcul

i) Top-down (de haut en bas) :

→ Calcul récursif : PCCH(n-1,t)

II. Programmation dynamique

d) Calcul

i) Top-down (de haut en bas) :

→ Calcul récursif : PCCH(n-1,t)

```
PPCH(k,v) :
    Si k=0 alors …
    Sinon :
        R= PPCH(k-1,v)
        soit u1,…,uz les prédécesseurs de v
        Pour i de 1 à z :
            T= PCCH(k-1,ui)+d(ui,v)
            Si T<R alors R←T
        Renvoyer R
```

Programmation dynamique

II. Programmation dynamique

d) Calcul

i) Top-down (de haut en bas) :

Problème : on va calculer plusieurs fois la même valeur (cf Fibonacci)
→ Non efficace.

Programmation dynamique

II. Programmation dynamique

d) Calcul

i) Top-down (de haut en bas) :

Problème : on va calculer plusieurs fois la même valeur (cf Fibonacci)
→ Non efficace.
→ Stocker les valeurs calculées : memo (memoïsation)

II. Programmation dynamique

d) Calcul

i) Top-down (de haut en bas) :

Problème : on va calculer plusieurs fois la même valeur (cf Fibonacci)
→ Non efficace.
→ Stocker les valeurs calculées : memo (memoïsation)

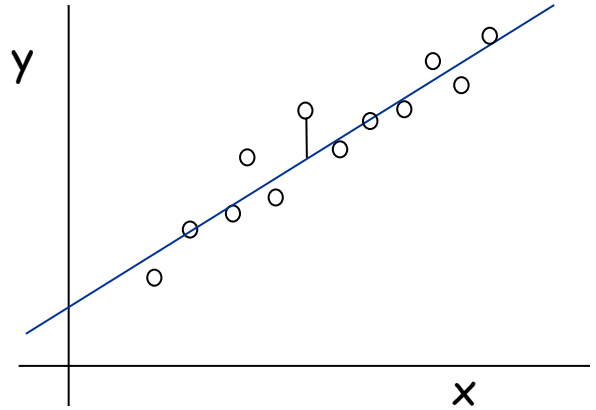
```
PPCH(k,v,memo) :
    Si (k,v) dans memo alors renvoyer memo(k,v).
    Si k=0 alors …
    Sinon :
        R= PCCH(k-1,v)
        soit u1,…,uz les prédécesseurs de v
        Pour i de 1 à z :
            T= PCCH(k-1,ui)+d(ui,v)
            Si T<R alors R←T
        Stocker R dans memo(k,v)
        (Renvoyer R)
```

Segmentation aux moindres carrés

Moindres carrés.

- Problème fondamental en statistique et analyse numérique.
- Étant donné n points dans le plan : $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Trouvez une droite $y = ax + b$ qui minimise la somme des erreurs quadratiques :

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Solution. Calcul $\Rightarrow$ L'erreur minimale est obtenue lorsque

$$a = \frac{n\sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2}, b = \frac{\sum_i y_i - a\sum_i x_i}{n}$$

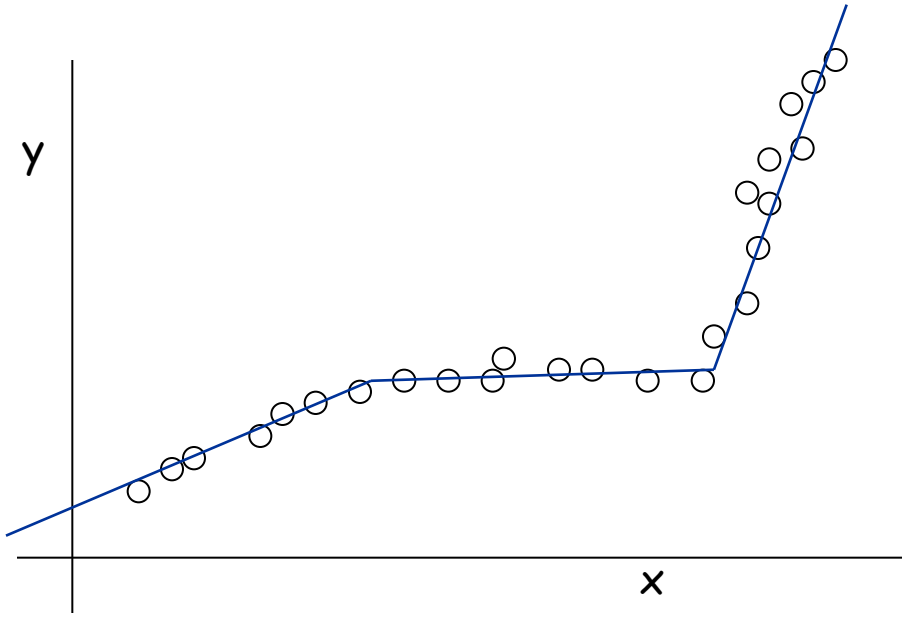
Segmentation aux moindres carrés

- Les points se trouvent approximativement sur une séquence de plusieurs segments de droite.
- Étant donné n points dans le plan $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ avec $x_1 < x_2 < \ldots < x_n$, trouver une séquence de droites qui minimise f(x).

Q. Quel est le choix raisonnable pour f(x) afin d'équilibrer précision et parcimonie ?

bon ajustement

nombre de droites



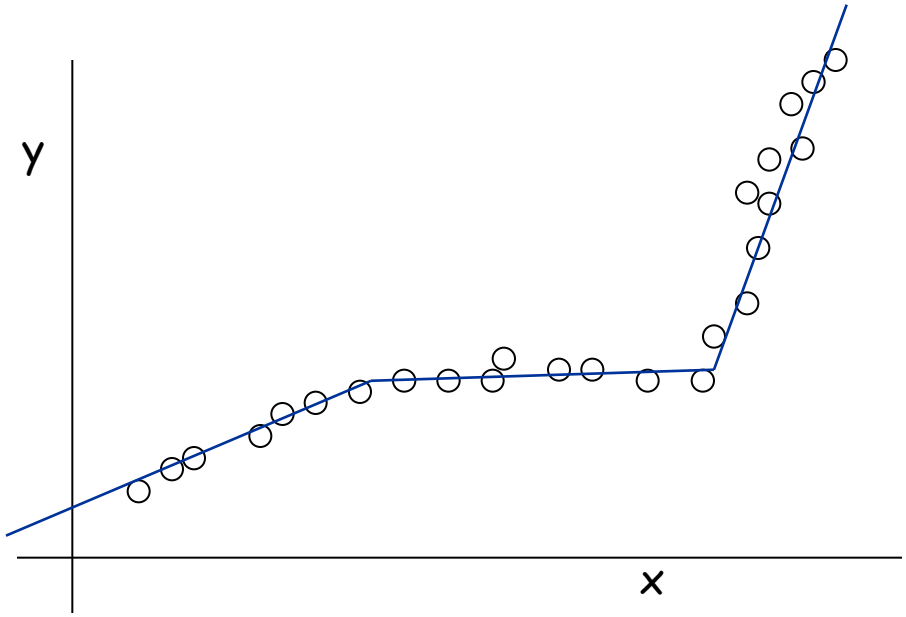
Segmentation aux moindres carrés

- Les points se trouvent approximativement sur une séquence de plusieurs segments de droite.
- Étant donné n points dans le plan $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ avec $x_1 < x_2 < \ldots < x_n$, trouver une séquence de droites qui minimise

  - la somme des sommes des erreurs quadratiques E dans chaque segment

  - le nombre de droites L

Fonction de compromis : $E + c\,L$, pour une constante $c > 0$.



Programmation Dynamique : Choix multiple

Notation.

- OPT(j) = coût minimum pour les points $p_1, p_{i+1}, \ldots, p_j$.
- e(i, j) = somme minimale des carrés pour les points $p_i, p_{i+1}, \ldots, p_j$.

Pour calculer OPT(j):

- Le dernier segment utilise des points $p_i, p_{i+1}, \ldots, p_j$ pour un certain i.
- Coût = e(i, j) + c + OPT(i-1).

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

Segmentation aux moindres carrés : Algorithme

```
INPUT: n, p_1,…,p_N , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error e_ij for
         the segment p_i,…, p_j

   for j = 1 to n
      M[j] = min_{1 ≤ i ≤ j} (e_ij + c + M[i-1])

   return M[n]
}
```

Temps de calcul. $O(n^3)$.

- Bottleneck = computing e(i, j) for $O(n^2)$ pairs, $O(n)$ per pair using previous formula.

Programmation dynamique

Conclusion :

→ Technique algorithmique permettant de parcourir efficacement un ensemble de solutions (partielles) parmi laquelle se trouve in fine une solution optimale.

→ Parfois gourmande en place mémoire

→ Autres exemples : alignement de séquences, sac-à-dos, planification, tomographie, …

→ Pourquoi le nom 'programmation dynamique' ?

Programmation dynamique

"Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. [...] You can imagine how he felt, then, about the term mathematical. Hence, I felt I had to do something to shield Wilson [...] from the fact that I was really doing mathematics […].

What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming".

I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying.

I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible.

Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities. »

Bellman, Eye of the Hurricane: An Autobiography (1984).