PRALG : séance 6

ÉCOLE NATIONALE DES PONTS ET CHAUSSÉES

IP PARIS

Programmation dynamique : Aspects pratiques et implémentation

Pascal Monasse/Renaud Marlet
Laboratoire LIGM-IMAGINE

(avec de nombreux emprunts à Leiserson, Rivest & Stein 2009)

Contenu du cours

- **Technique très générale** pour résoudre certains pbs d'**optimisation discrète** en **temps polynomial**
  - pas d'énumération d'un nb exponentiel de possibilités
  - minimisation de sommes de fonctions croissantes sous contrainte
- Deux exemples détaillés
  - ligne d'assemblage, produit de plusieurs matrices
- Généralisation
- **Aspects informatiques** ($\neq$ cours de Recherche Opérat.)
- TP

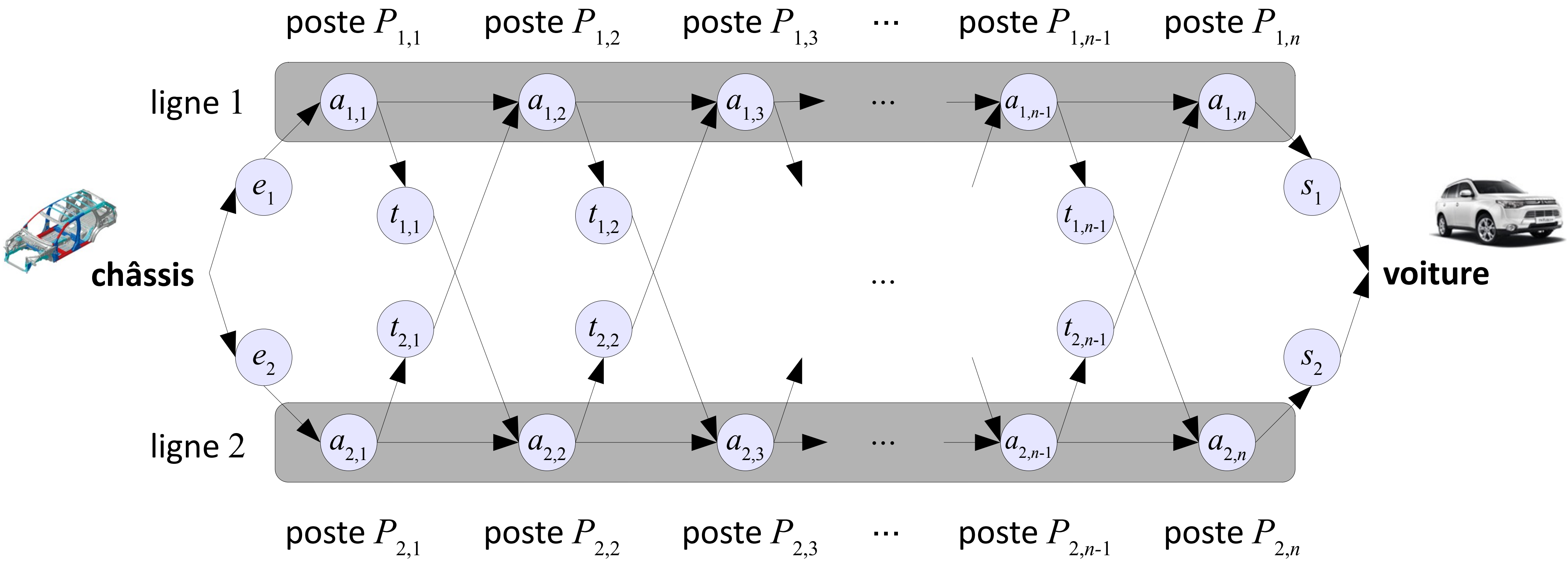
Notation (complexité)

- Borne supérieure asymptotique
  - $O(g(n)) = \{ f \mid \exists\, n_0, c > 0 \text{ tq } \forall\, n \geq n_0, 0 \leq f(n) \leq c\, g(n) \}$
- Borne inférieure asymptotique
  - $\Omega(g(n)) = \{ f \mid \exists\, n_0, c > 0 \text{ tq } \forall\, n \geq n_0, 0 \leq c\, g(n) \leq f(n) \}$
- Encadrement asymptotique
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
  - $\Theta(g(n)) = \{ f \mid \exists\, n_0, c_1, c_2 > 0 \text{ tq } \forall\, n \geq n_0,$
    $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$

Exemple 1 : ordonnancement d'une ligne d'assemblage

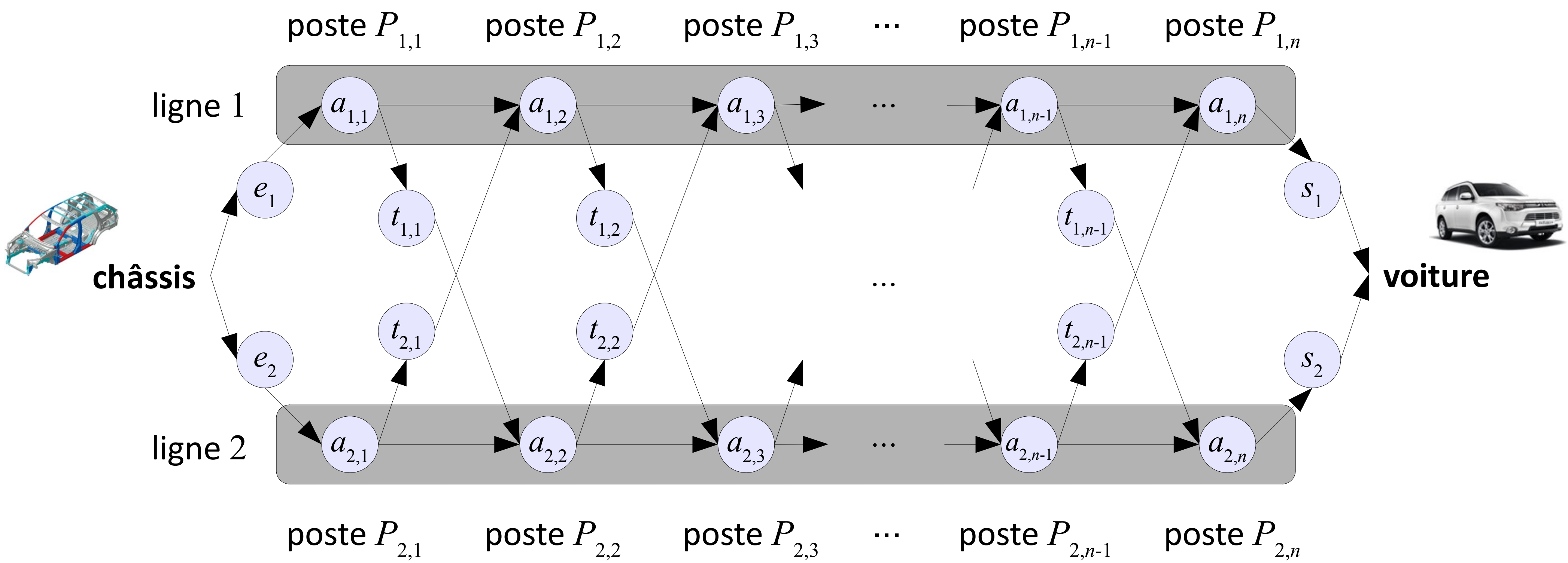
Lignes d'assemblage différentes
(ex. constituées à différents moments avec différentes technologies)

- e_i : temps d'acheminement à l'**e**ntrée de la ligne i
- $a_{i,j}$: temps requis pour effectuer l'**a**ssemblage du poste $P_{i,j}$
- $t_{i,j}$: temps de **t**ransfert du poste $P_{i,j}$ ligne i à la ligne $3-i$ (= à l'autre ligne)
- s_i : temps d'acheminement à la **s**ortie depuis la ligne i



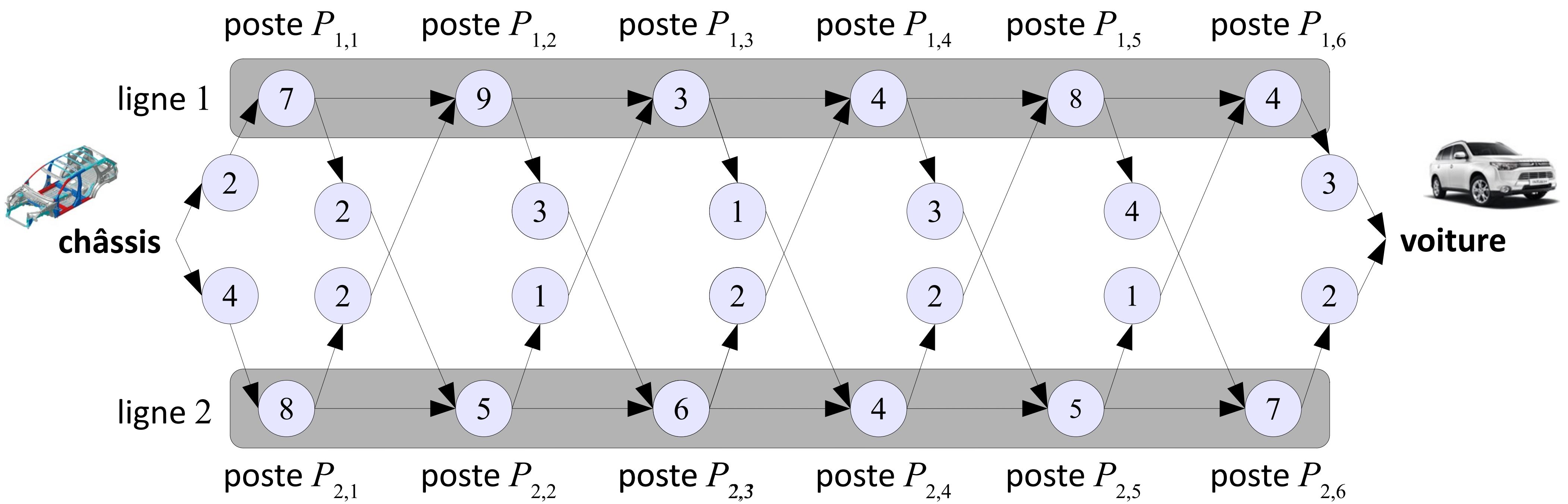
Temps d'assemblage ordinaire

- Temps d'assemblage avec la ligne i seulement :
  - $T_i = e_i + \sum_{j=1}^{n} a_{i,j} + s_i$



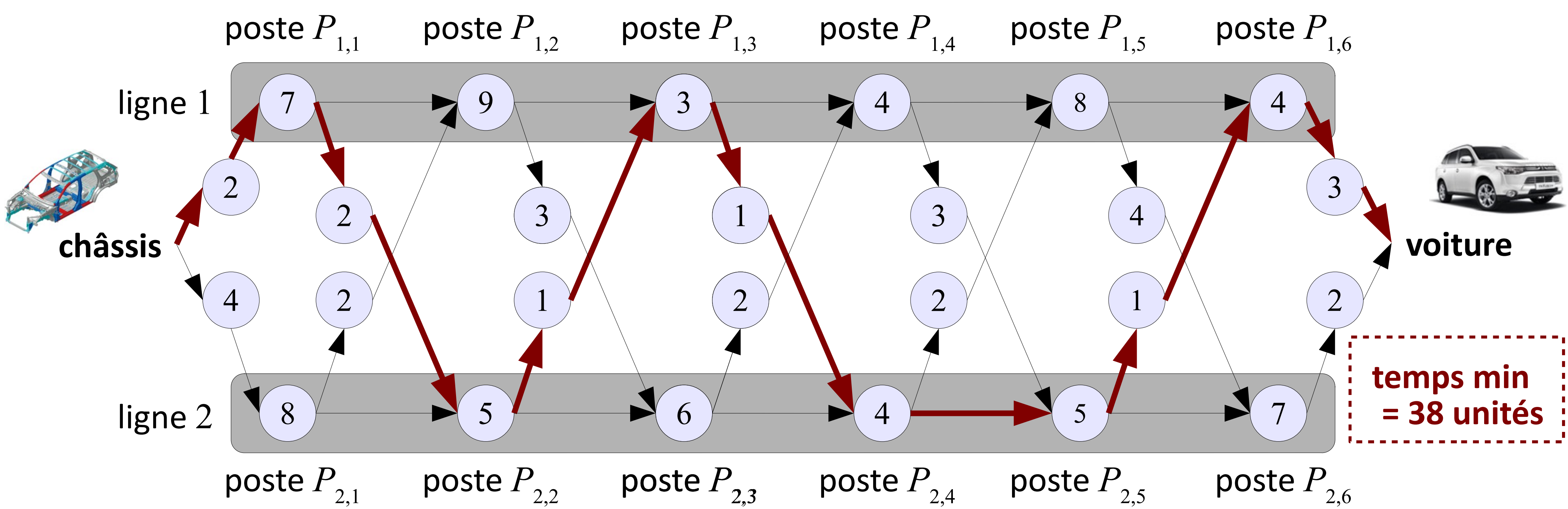
Cas d'une commande spéciale urgente

- Transferts de ligne pour utiliser les postes rapides
- Quel est le temps d'assemblage le plus court ?
  (= quel est le trajet le plus rapide ?)



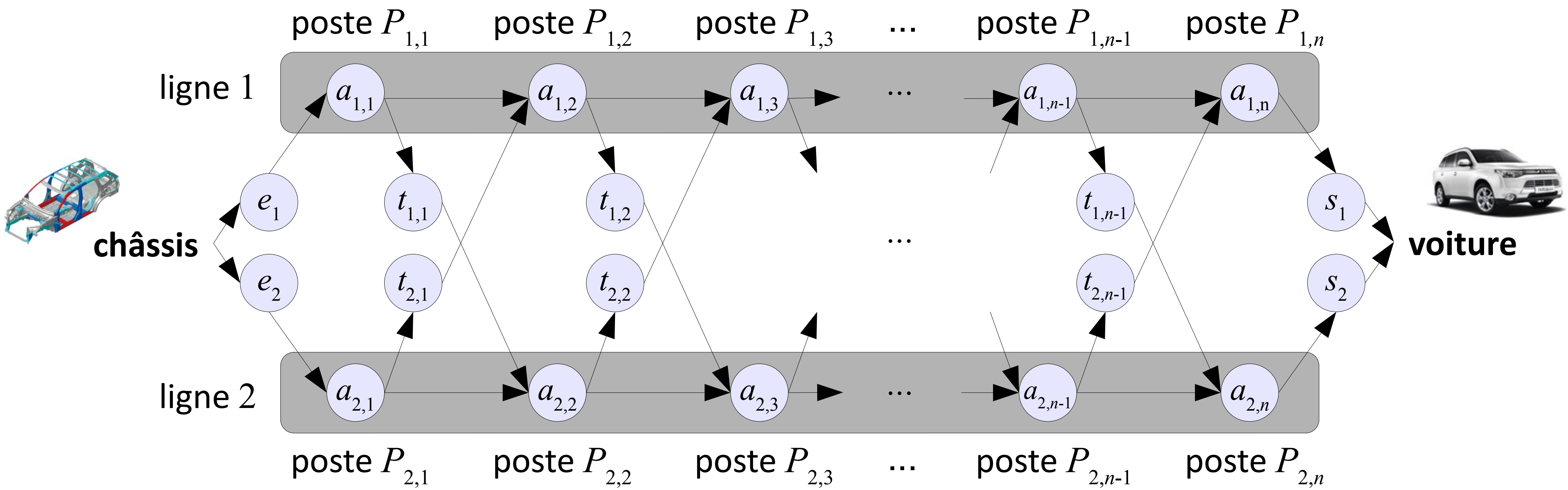
Cas d'une commande spéciale urgente

- Transferts de ligne pour utiliser les postes rapides
- Quel est le temps d'assemblage le plus court ?
  (= quel est le trajet le plus rapide ?)



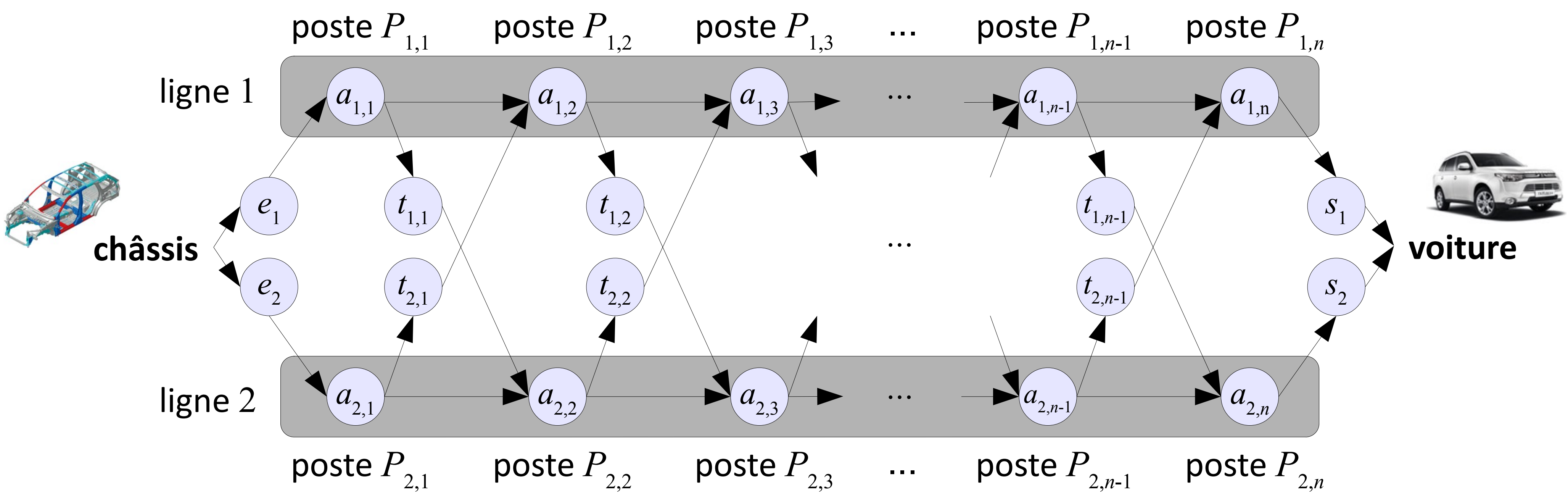
Cas d'une commande spéciale urgente

- Transferts de ligne pour utiliser les postes rapides
- S'il y a n postes, combien de trajets faut-il examiner ?



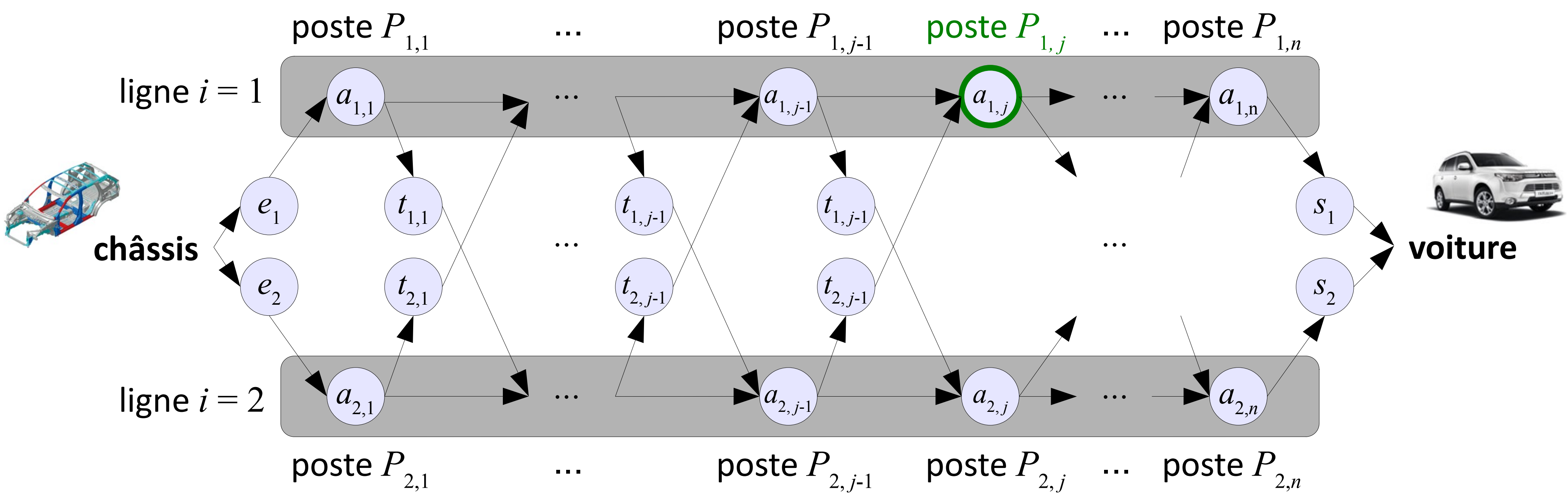
Cas d'une commande spéciale urgente

- Transferts de ligne pour utiliser les postes rapides
- S'il y a n postes, combien de trajets faut-il examiner ?
  - nombre de trajets différents = 2^n
  - ☛ recherche exhaustive **inapplicable en pratique**



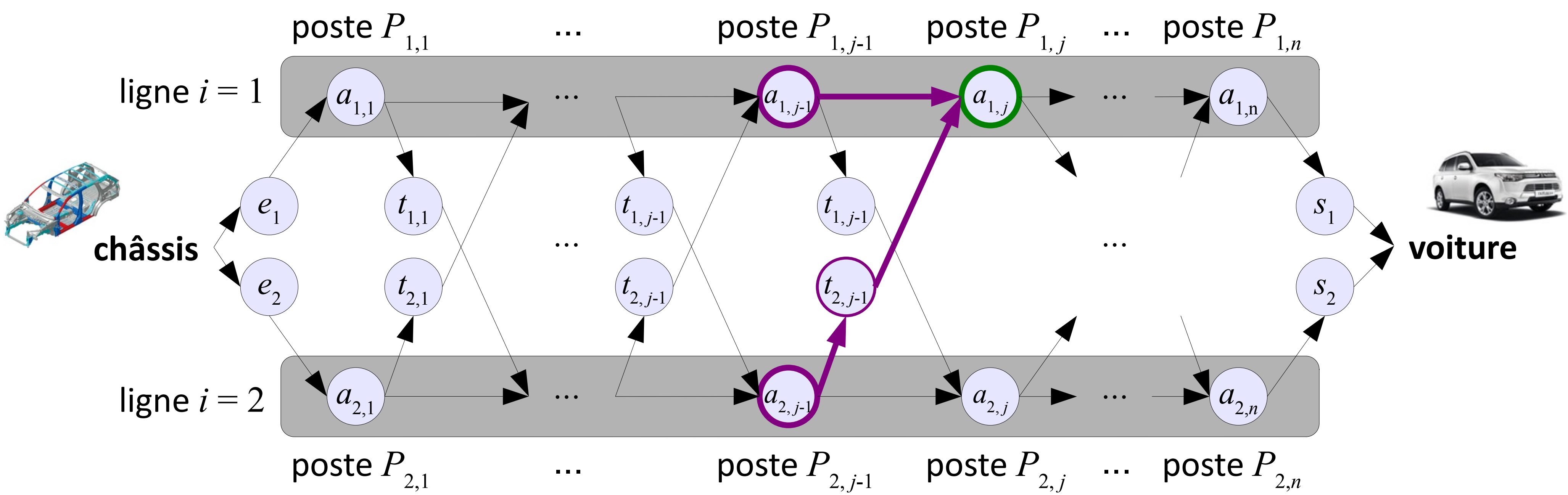
Chemins intermédiaires les plus rapides

- **Question :** quel est le chemin le plus rapide jusqu'à $P_{i,j}$?



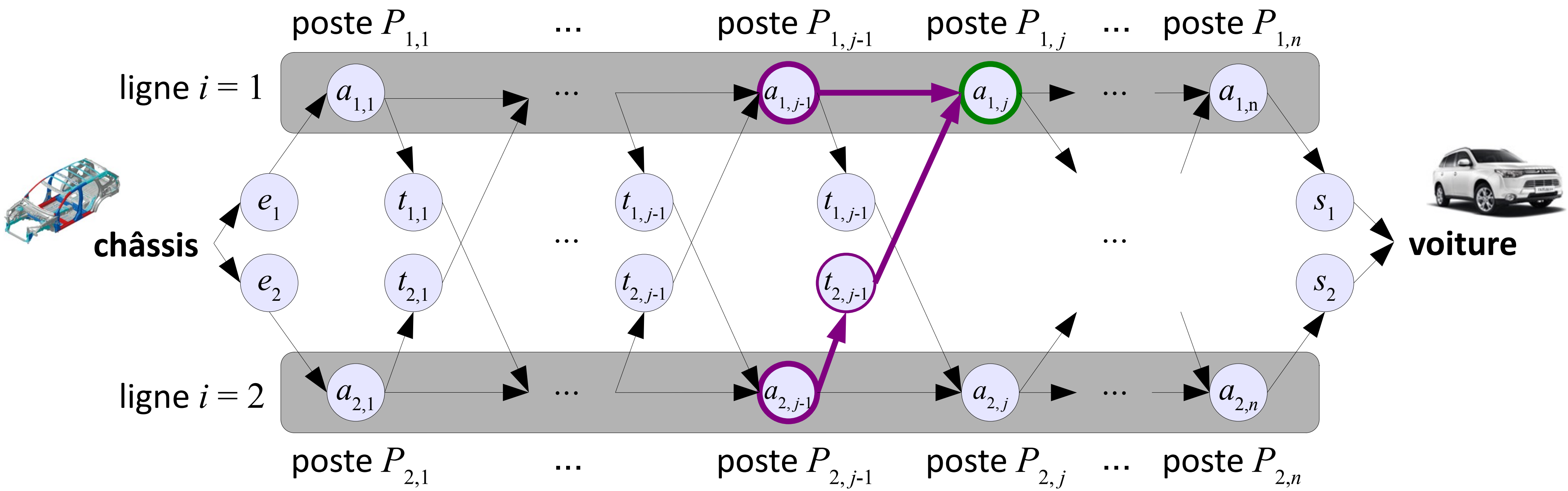
Chemins intermédiaires les plus rapides

- **Question :** quel est le chemin le plus rapide jusqu'à $P_{i,j}$?
  - si $j = 1$, un seul chemin
  - si $j > 1$, deux cas : le châssis vient de $P_{1,\, j\text{-}1}$ ou bien de $P_{2,\, j\text{-}1}$
- **Question :** son sous-chemin jusqu'à $P_{i',\, j\text{-}1}$ est-il aussi le plus rapide ?



Chemins intermédiaires les plus rapides

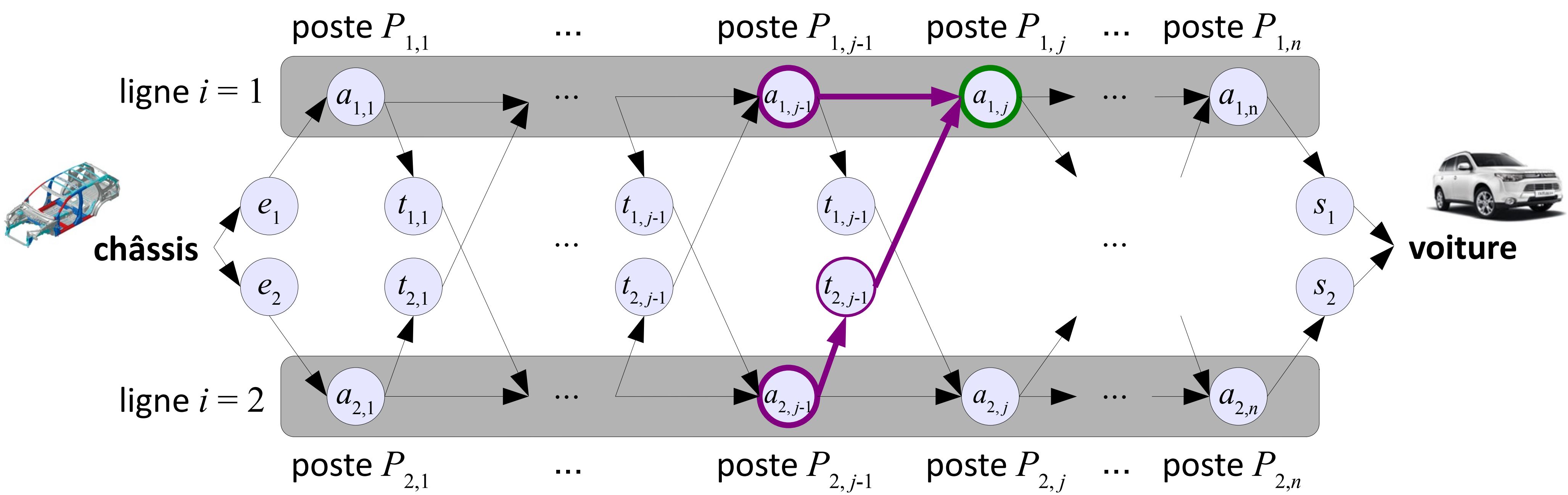
- **Question :** quel est le chemin le plus rapide jusqu'à $P_{i,j}$?
  - si $j = 1$, un seul chemin
  - si $j > 1$, deux chemins : le châssis vient de $P_{1,j\text{-}1}$ ou bien de $P_{2,j\text{-}1}$
- **Propriété :** si le chemin le plus rapide jusqu'à $P_{i,j}$ passe par $P_{i',j\text{-}1}$ alors son sous-chemin jusqu'à $P_{i',j\text{-}1}$ est aussi le plus rapide à $P_{i',j\text{-}1}$



Chemins intermédiaires les plus rapides

- **Question :** quel est le chemin le plus rapide jusqu'à $P_{i,j}$?
  - si $j = 1$, un seul chemin
  - si $j > 1$, deux chemins : le châssis vient de $P_{1,\, j\text{-}1}$ ou bien de $P_{2,\, j\text{-}1}$
- **Propriété :** si le chemin le plus rapide jusqu'à $P_{i,\, j}$ passe par $P_{i',\, j\text{-}1}$ alors son sous-chemin jusqu'à $P_{i',\, j\text{-}1}$ est aussi le plus rapide à $P_{i',\, j\text{-}1}$

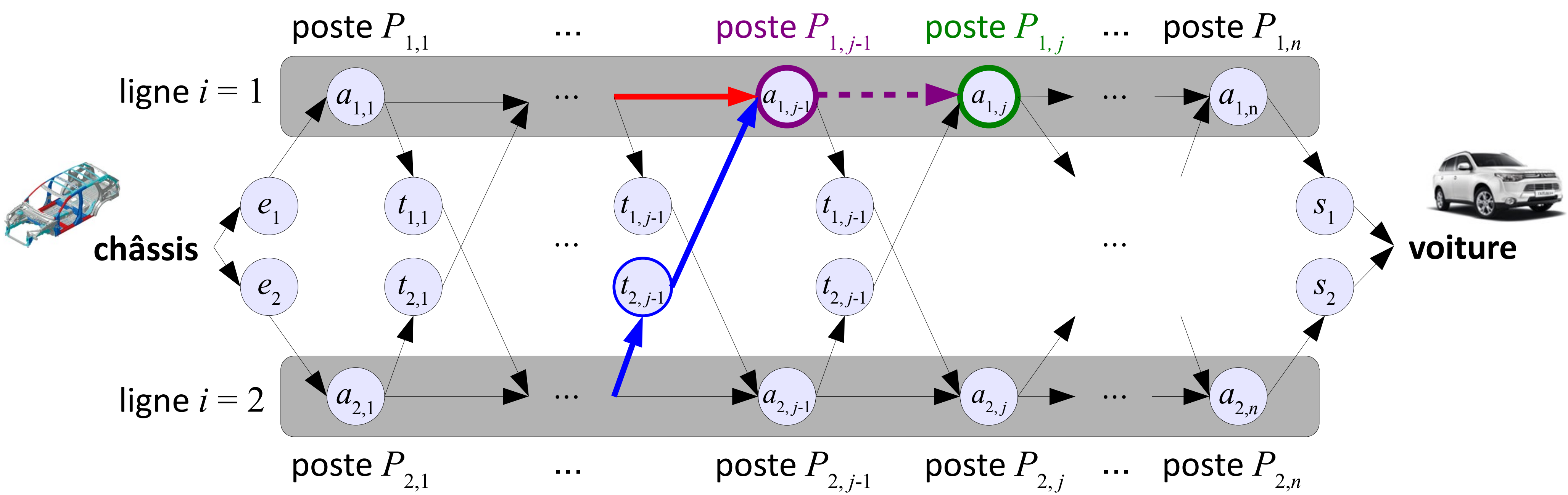
preuve ?



Chemins intermédiaires les plus rapides

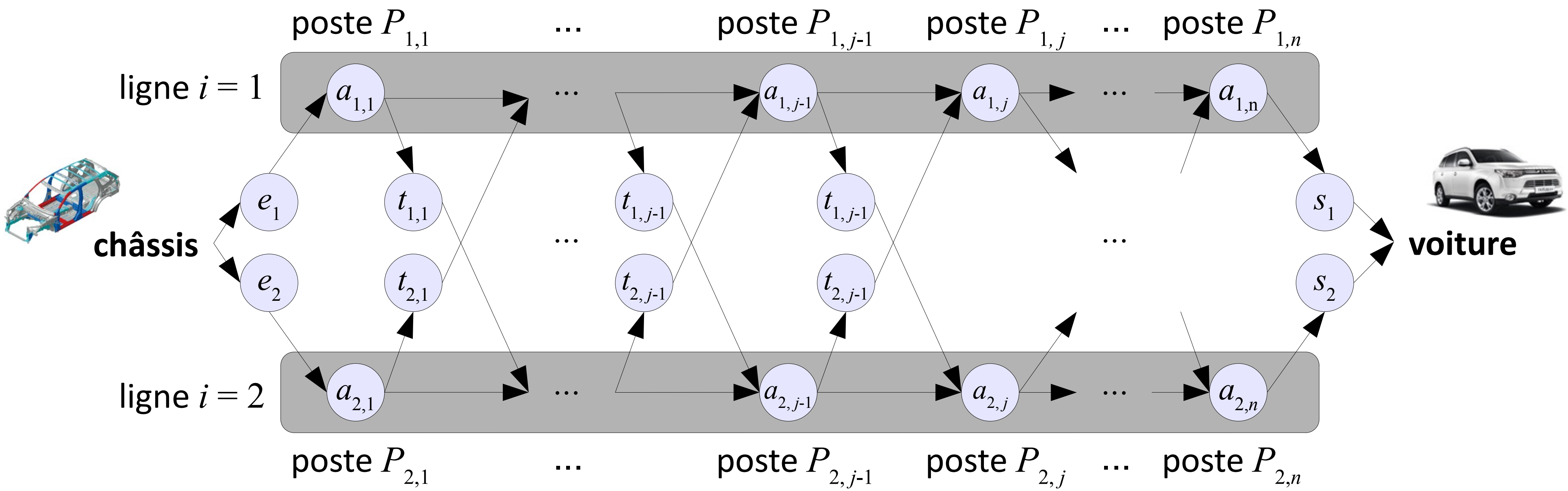
preuve par l'absurde

- Hypothèse : le chemin C le plus rapide jusqu'à $P_{i,\,j}$ passe par $P_{i',\,j\text{-}1}$ mais son sous-chemin C' jusqu'à $P_{i',\,j\text{-}1}$ n'est pas le plus rapide
- Alors un autre sous-chemin C'' jusqu'à $P_{i',\,j\text{-}1}$ est le plus rapide, mais alors C'' + la transition de $P_{i',\,j\text{-}1}$ à $P_{i,\,j}$ est plus rapide que C : contrad!
- Donc C' est le chemin le plus rapide jusqu'à $P_{i',\,j\text{-}1}$



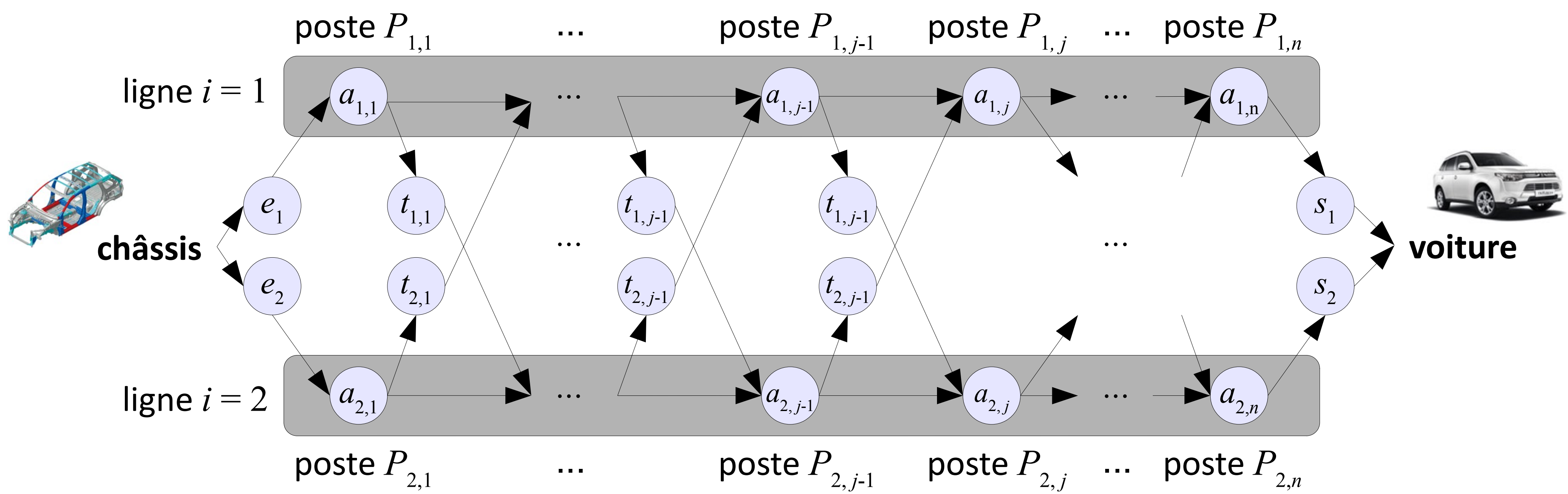
Propriété de sous-structure optimale

- **Définition :** un problème a une **sous-structure optimale** ssi une solution optimale du problème contient une solution optimale pour les sous-problèmes [*propriété de Bellman*]
- **Exemple :** si le chemin le plus rapide jusqu'à $P_{i,j}$ passe par $P_{i',j-1}$ alors le sous-chemin jusqu'à $P_{i',j-1}$ est aussi le plus rapide



Méthode : solution au problème général via les solutions aux sous-problèmes

- Considérer le cas de $j \leq n$: ici, chemin le plus rapide jusqu'à $P_{1,j}$?
  - chemin le plus rapide jusqu'à $P_{1,j\text{-}1}$ puis direct à $P_{1,j}$ **ou bien**
  - chemin le plus rapide jusqu'à $P_{2,j\text{-}1}$ puis transfert sur ligne 1
- Idem, symétriquement, pour $P_{2,j}$
- Puis prendre $j = n$



Solution générale

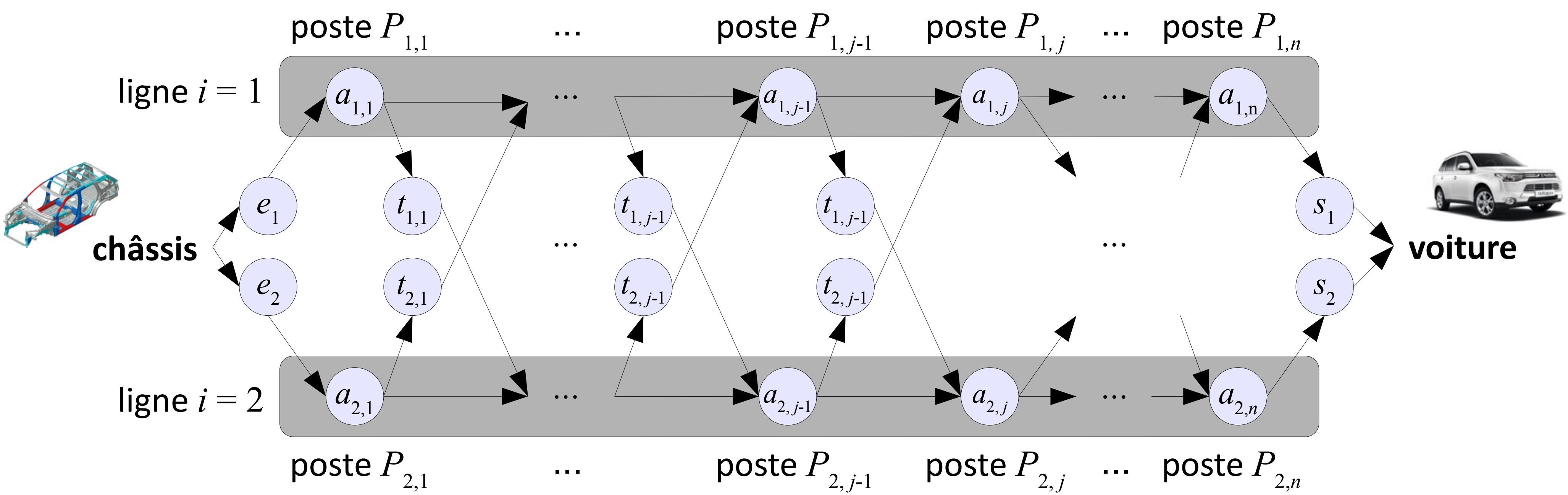
- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action $a_{i,j}$

Équation de Bellman

  - $r_{i,1} = e_i + a_{i,1}$ si $j = 1$
  - $r_{i,j} = \min(r_{i,j\text{-}1}, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 1$ (avec $k = 3 - i$: autre ligne que i)

    $[i = 1 \Rightarrow k = 2 \;|||\; i = 2 \Rightarrow k = 1]$

- r^* : temps minimum de parcours total
  - $r^* = \min(r_{1,n} + s_1 , r_{2,n} + s_2)$

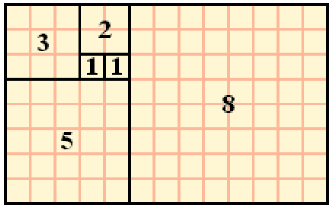


Intermède

- Leonardo Fibonacci
  - mathématicien italien (v. 1175 – v. 1250)
- Notamment connu pour
  - l'introduction en Europe de l'écriture décimale positionnelle (système de numération indo-arabe)
  - la suite numérique

■ $$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \text{ pour } n \geq 0 \end{cases}$$

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	…
0	1	1	2	3	5	8	13	21	…



■ $$F_n \approx \frac{1}{\sqrt{5}} \varphi^n \text{ avec } \varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618 \quad \text{(nombre d'or)}$$

Comment calculer F_n en pratique ?

Légende : codage direct / gestion spécifique / attention oubli fréquent

- Version **récursive**

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

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{n+2} = F_{n+1} + F_n \quad \text{pour } n \geq 0$$

Variante :

$F_n = n$ si $n \leq 1$

$F_n = F_{n-1} + F_{n-2}$ sinon

- Version **itérative naïve 1**

```
int fib(int n) {
  if (n <= 1) return n;
  int *F = new int[n+1]; // Allocation dynamique explicite
  F[0] = 0; F[1] = 1;
  for(int i = 2; i <= n; i++)
    F[i] = F[i-1] + F[i-2];
  int Fn = F[n];
  delete [] F;             // Libération explicite de l'espace mémoire
  return Fn;
}
```

F_0	F_1	F_2	F_3	F_4	F_5	F_6	...
0	1	1	2	3	5	8	...

Comment calculer F_n en pratique ?

Légende : codage direct / gestion spécifique / attention oubli fréquent

- Version **récursive**

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

$F_0 = 0$
$F_1 = 1$
$F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$

Variante :
$F_n = n$ si $n \leq 1$
$F_n = F_{n-1} + F_{n-2}$ sinon

- Version **itérative naïve 2** (un poil moins efficace, mais plus lisible)

```
int fib(int n) {
  if (n <= 1) return n;
  vector<int> F(n+1);     // Allocation dynamique implicite
  F[0]=0;
  F[1]=1;
  for(int i = 2; i <= n; i++)
    F[i] = F[i-1] + F[i-2];
  return F[n];            // Libération implicite de l'espace mémoire
}
```

F_0	F_1	F_2	F_3	F_4	F_5	F_6	...
0	1	1	2	3	5	8	...

Comment calculer F_n en pratique ?

Légende : codage direct / gestion spécifique / attention oubli fréquent

- Version **récursive**

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

$F_0 = 0$
$F_1 = 1$
$F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$

Variante :
$F_n = n$ si $n \leq 1$
$F_n = F_{n-1} + F_{n-2}$ sinon

- Version **itérative naïve 2** (un poil moins efficace, mais plus lisible)

```
int fib(int n) {
  if (n <= 1) return n;
  vector<int> F(n+1);    // Allocation dynamique implicite
  F[0]=0;
  F[1]=1;
  for(int i = 2; i <= n; i++)
    F[i] = F[i-1] + F[i-2];
  return F[n];           // Libération implicite de l'espace mémoire
}
```

F_0	F_1	F_2	F_3	F_4	F_5	F_6	...
0	1	1	2	3	5	8	...

Combien cela fait-il de calculs ?

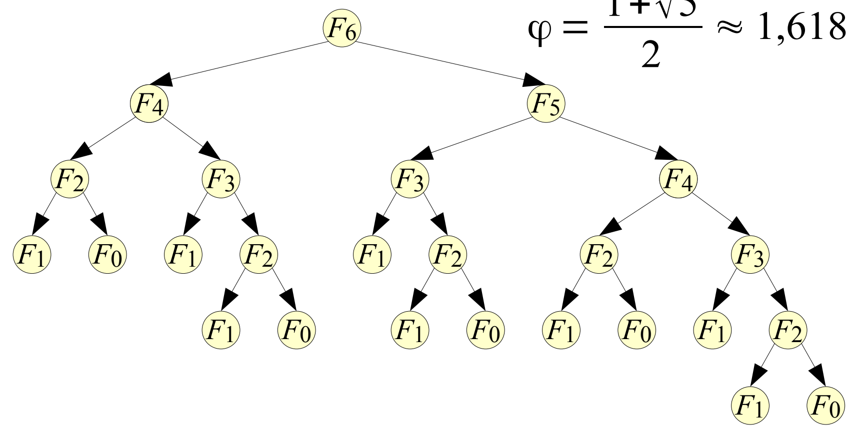
Comment calculer F_n en pratique ?

Légende : codage direct / gestion spécifique / attention oubli fréquent

- Version **récursive**

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

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618$$

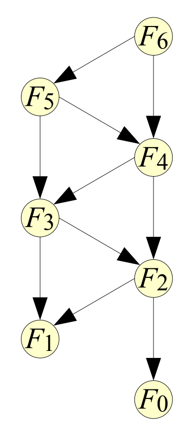


Nb d'appels dans `fib` $(n) = F_{n-1} = \Theta(\varphi^n)$

Nb d'additions dans `fib` $(n) = F_{n-2} = \Theta(\varphi^n)$

- Version **itérative naïve 2**

```
int fib(int n) {
  if (n <= 1) return n;
  vector<int> F(n+1);
  F[0]=0;
  F[1]=1;
  for(int i = 2; i <= n; i++)
    F[i] = F[i-1] + F[i-2];
  return F[n];
}
```



Nb d'itérations = nb `F`-additions pour `fib` $(n) = n-1 = \Theta(n)$

Comment calculer F_n en pratique ?

Légende : codage direct / gestion spécifique / attention oubli fréquent

- Version **itérative naïve 2**

```
int fib(int n) {
  if (n <= 1) return n;
  vector<int> F(n+1);
  F[0]=0; F[1]=1;
  for(int i = 2; i <= n; i++)
    F[i] = F[i-1] + F[i-2];
  return F[n];
}
```

- Version **itérative sans allocation**

```
int fib(int n) {
  int Fn = n, Fn1 = 1, Fn2 = 0;
  for(int i = 2; i <= n; i++) {
    Fn = Fn1 + Fn2;
    Fn2 = Fn1;
    Fn1 = Fn; }
  return Fn;
}
```

F_0	F_1	F_2	F_3	F_4	F_5	F_6	...
0	1	1	2	3	5	8	...

$F_0 = 0$

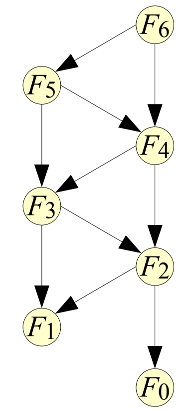
$F_1 = 1$

$F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$

Variante :

$F_n = n$ si $n \leq 1$

$F_n = F_{n-1} + F_{n-2}$ sinon



Nb d'itérations = nb `F`-additions pour `fib`(n) $= n-1 = \Theta(n)$

Comment calculer F_n en pratique ?

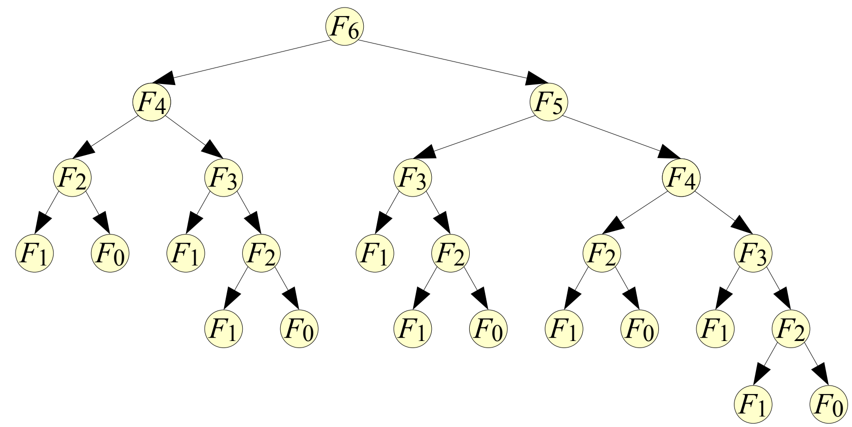
Légende : codage direct / gestion spécifique / attention oubli fréquent

- Version **récursive compacte**

```
int fib(int n) {
  return (n <= 1) ?
    n : fib(n-1)+fib(n-2);
}
```

Nb d'appels dans `fib(n)` : $F_{n\text{-}1} = \Theta(\varphi^n)$

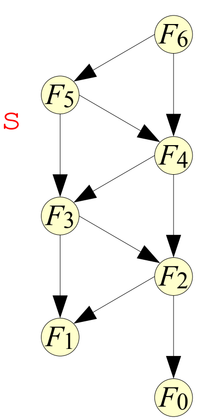
Nb d'additions dans `fib(n)` : $F_{n\text{-}2} = \Theta(\varphi^n)$



C, C++ : *condition ? valeur_si_vrai : valeur_si_faux*

- Version **récursive mémoïsée**
  = avec « cache » des résultats déjà calculés

```
vector<int> F; // semi-statique
int fib(int n) {
  if (F.size() < n+1)
    F.resize(n+1,-1); // Valeur par défaut=-1 pour les nouvelles cases
  if (F[n] == -1)     // Si F[n] pas encore calculé
    F[n] = n<=1 ? n : fib(n-1)+fib(n-2);
  return F[n];
}
```



Nb d'appels dans `fib(n)` : au plus $n+1 = \mathrm{O}(n)$

Nb de `F`-additions : au plus $n-1 = \mathrm{O}(n)$

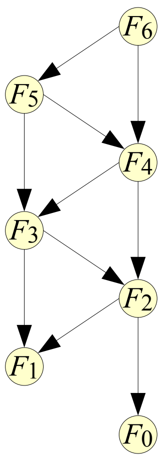
Comment calculer F_n en pratique ?

Légende : codage direct / gestion spécifique / attention oubli fréquent

- Version **itérative naïve 2**

Nb d'itérations = nb F-additions : $n-1=\Theta(n)$
Nb d'allocations : 1 par appel

```
int fib(int n) {
  if (n <= 1) return n;
  vector<int> F(n+1); // Allocation dynamique
  F[0]=0; F[1]=1;
  for(int i = 2; i <= n; i++)
    F[i] = F[i-1]+F[i-2];
  return F[n]; }
```



- Version **itérative mémoïsée**

```
vector<int> F = {0,1}; // C++11
int fib(int n) {
  int s = F.size();            // Nombre de valeurs précalculées et mémorisées
  if (s < n+1) F.resize(n+1); // S'assurer que la table va jusqu'à F[n]
  for(int i = s; i <= n; i++) // Boucle que si n > s, pour compléter F
    F[i] = F[i-1]+F[i-2];
  return F[n];
}
```

Nb d'itérations = nb F-additions : au plus $n-1=O(n)$
Nb d'allocations : moins de 1 par appel

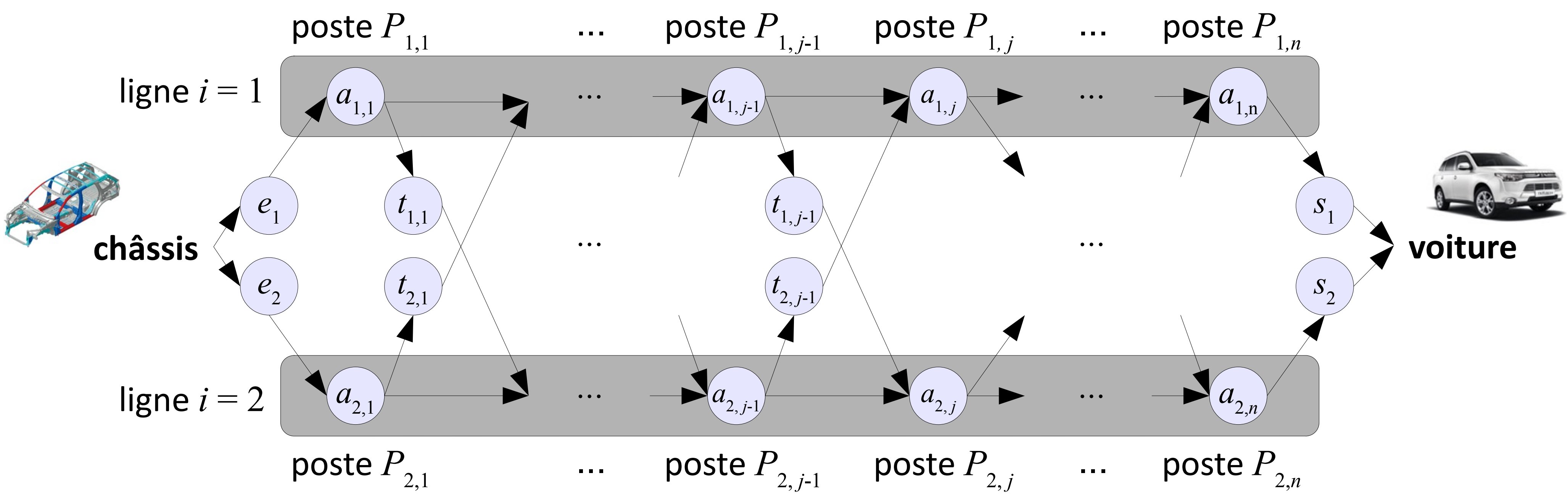
Solution générale

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action

Équation de Bellman

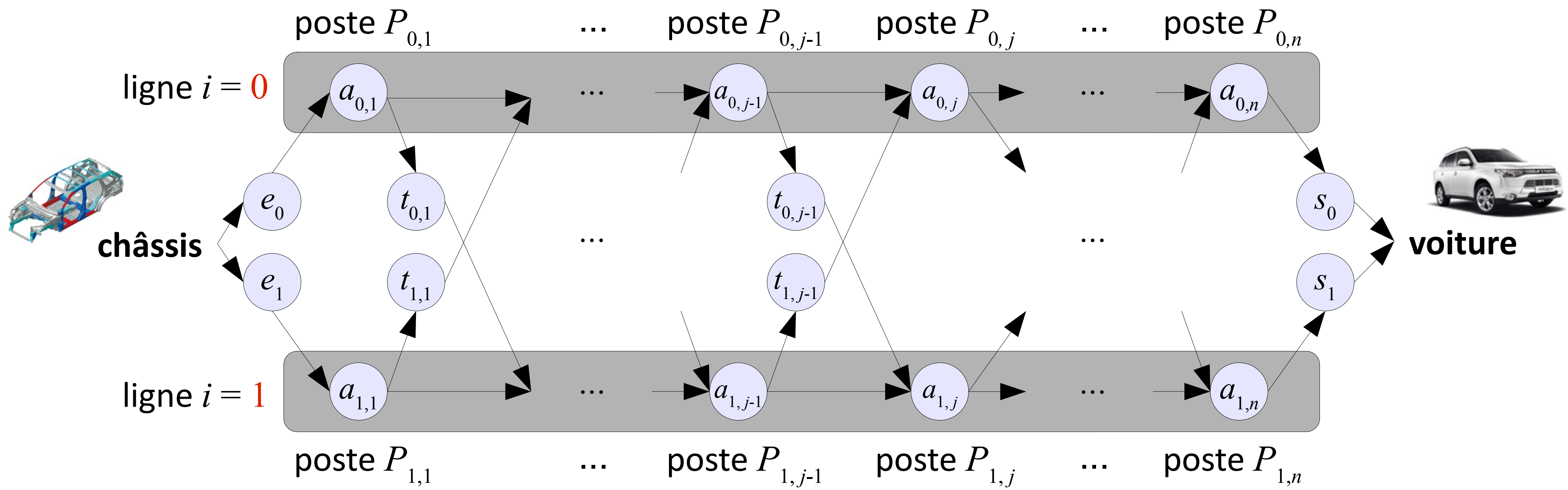
  - $r_{i,1} = e_i + a_{i,1}$ si $j = 1$
  - $r_{i,j} = \min(r_{i,j\text{-}1}\,,\, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 1$ (avec $k = 3 - i$: autre ligne que i)
[i = 1 ⇒ k = 2 ||| i = 2 ⇒ k = 1]

- r^* : temps optimal de parcours total
  - $r^* = \min(r_{1,n} + s_1\,,\, r_{2,n} + s_2)$



Adaptation des indices pour correspondre aux tableaux de C/C++

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}\,,\, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i) [$i = 0 \Rightarrow k = 1$ ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0\,,\, r_{1,n\text{-}1} + s_1)$



Solution récursive directe (≈ exhaustive ☹)

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}\,,\, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i) [i = 0 ⇒ k = 1 ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0\,,\, r_{1,n\text{-}1} + s_1)$

Complexité : ???

```
float r(int i,int j) {
  if (j==0) return e[i] + a[i][0];
  int k = 1-i;
  return min(r(i,j-1), r(k,j-1)+t[k][j-1]) + a[i][j];
}
ropt = min(r(0,n-1)+s[0], r(1,n-1)+s[1]);
```

Solution récursive directe (≈ exhaustive ☹)

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}\, ,\, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i) [i = 0 ⇒ k = 1 ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0\, ,\, r_{1,n\text{-}1} + s_1)$

Complexité : $\Theta(2^n)$

```
float r(int i,int j) {
  if (j==0) return e[i] + a[i][0];
  int k = 1-i;
  return min(r(i,j-1), r(k,j-1)+t[k][j-1]) + a[i][j];
}
ropt = min(r(0,n-1)+s[0], r(1,n-1)+s[1]);
```

Solution récursive directe
Variante spécialisée (≈ exhaustive ☹)

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i)

    [$i = 0 \Rightarrow k = 1$ ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0, r_{1,n\text{-}1} + s_1)$

```
float r0(int j){
  if (j==0) return e0+a0[0];
  else      return min(r0(j-1), r1(j-1)+t1[j-1]) + a0[j];
}
float r1(int j) {
  if (j==0) return e1+a1[0];
  else      return min(r1(j-1), r0(j-1)+t0[j-1]) + a1[j];
}
ropt = min(r0(n-1)+s0, r1(n-1)+s1);
```

Complexité : $\Theta(2^n)$

Pour la linéarité, comment faire ?

Solution récursive directe
Variante spécialisée (≈ exhaustive ☹)

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i)

    [$i = 0 \Rightarrow k = 1$ ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0, r_{1,n\text{-}1} + s_1)$

Complexité : $\Theta(2^n)$

```
float r0(int j){
  if (j==0) return e0+a0[0];
  else      return min(r0(j-1), r1(j-1)+t1[j-1]) + a0[j];
}
float r1(int j) {
  if (j==0) return e1+a1[0];
  else      return min(r1(j-1), r0(j-1)+t0[j-1]) + a1[j];
}
ropt = min(r0(n-1)+s0, r1(n-1)+s1);
```

Pour la linéarité, il faudrait mémoïser

Solution itérative avec mémorisation (efficace ☺)

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}\,,\; r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i)

    $[i = 0 \Rightarrow k = 1 \,|||\, i = 1 \Rightarrow k = 0]$
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0\,,\; r_{1,n\text{-}1} + s_1)$

Complexité : ???

```
float *r[2];
for (int i=0; i <= 1; i++) {
  r[i] = new float[n];
  r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++)
  for (int i=0,k=1; i <= 1; i++,k=1-i)
    r[i][j] = min(r[i][j-1], r[k][j-1]+t[k][j-1]) + a[i][j];
ropt = min(r[0][n-1]+s[0], r[1][n-1]+s[1]);
delete[] r[0]; delete[] r[1];
```

Solution itérative avec mémorisation (efficace ☺)

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}\,,\ r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i) [$i = 0 \Rightarrow k = 1$ ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0\,,\ r_{1,n\text{-}1} + s_1)$

Complexité : $\Theta(n)$

```
float *r[2];
for (int i=0; i <= 1; i++) {
  r[i] = new float[n];
  r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++)
  for (int i=0,k=1; i <= 1; i++,k=1-i)
    r[i][j] = min(r[i][j-1], r[k][j-1]+t[k][j-1]) + a[i][j];
ropt = min(r[0][n-1]+s[0], r[1][n-1]+s[1]);
delete[] r[0]; delete[] r[1];
```

Solution itérative avec mémorisation Variante spécialisée (efficace ☺)

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}, r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i) [$i = 0 \Rightarrow k = 1$ ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0, r_{1,n\text{-}1} + s_1)$

Complexité : $\Theta(n)$

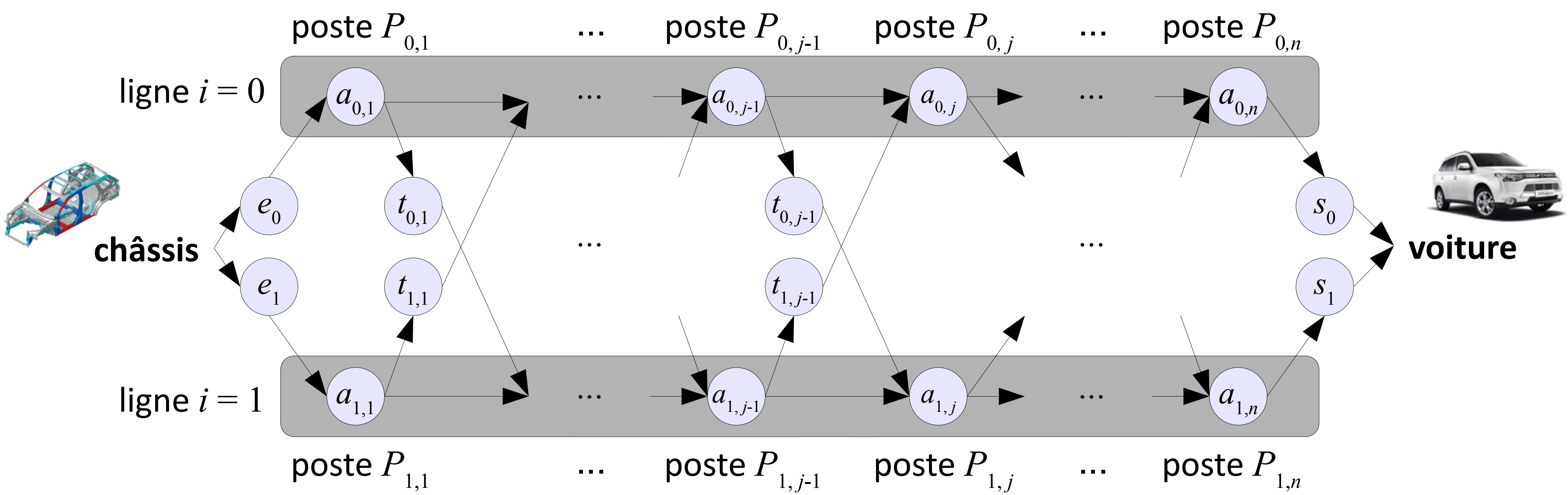
```
float *r0 = new float[n], *r1 = new float[n];
r0[0] = e0+a0[0];
r1[0] = e1+a1[0];
for (int j=1; j < n; j++) {
  r0[j] = min(r0[j-1], r1[j-1]+t1[j-1]) + a0[j];
  r1[j] = min(r1[j-1], r0[j-1]+t0[j-1]) + a1[j];
}
ropt = min(r0[n-1]+s0, r1[n-1]+s1);
delete[] r0; delete[] r1;
```

Solution générale

- $r_{i,j}$: temps le plus court pour aller au poste $P_{i,j}$ et y faire l'action
  - $r_{i,0} = e_i + a_{i,0}$ si $j = 0$
  - $r_{i,j} = \min(r_{i,j\text{-}1}\,,\ r_{k,j\text{-}1} + t_{k,j\text{-}1}) + a_{i,j}$ si $j > 0$ (avec $k = 1 - i$: autre ligne que i)

    [$\mathrm{i} = 0 \Rightarrow \mathrm{k} = 1$ ||| $i = 1 \Rightarrow k = 0$]
- r^* : temps optimal de parcours total
  - $r^* = \min(r_{0,n\text{-}1} + s_0\,,\ r_{1,n\text{-}1} + s_1)$

Ça dit **le temps le plus court** jusqu'en $P_{i,j}$
pas le chemin le plus court jusqu'en $P_{i,j}$



Retrouver une solution optimale ?

```
float *r[2];
for (int i=0; i <= 1; i++) {

  r[i] = new float[n]; r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++) { // ∀ poste
  for (int i=0,k=1; i <= 1 ; i++,k=1-i) // ∀ ligne

    r[i][j] =       min( r[i][j-1],          // Si on vient … de i

                         r[k][j-1]+t[k][j-1])+a[i][j];//… de k
}

ropt =    min( r[0][n-1]+s[0],

               r[1][n-1]+s[1] );
```

Retrouver une solution optimale : mémoriser au vol

- `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$

```
float *r[2]; int *l[2];
for (int i=0; i <= 1; i++) {
  l[i] = new int[n];   l[i][0] = i;
  r[i] = new float[n]; r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++) { // ∀ poste
  for (int i=0,k=1; i <= 1 ; i++,k=1-i) // ∀ ligne

    r[i][j] =      min( r[i][j-1],          // Si on vient … de i

                        r[k][j-1]+t[k][j-1])+a[i][j];//… de k
}

ropt=     min( r[0][n-1]+s[0],

               r[1][n-1]+s[1] );
```

Retrouver une solution optimale : mémoriser au vol

- `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$

```
float *r[2]; int *l[2];
for (int i=0; i <= 1; i++) {
  l[i] = new int[n];    l[i][0] = i;
  r[i] = new float[n]; r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++) { // ∀ poste
  for (int i=0,k=1; i <= 1 ; i++,k=1-i) // ∀ ligne

    r[i][j] =        min( r[i][j-1],            // Si on vient … de i

                          r[k][j-1]+t[k][j-1])+a[i][j];//… de k
}

ropt=      min( r[0][n-1]+s[0],

                r[1][n-1]+s[1] );
```

X = min(Y,Z);
⇔
if (Y < Z)
  X = Y;
else
  X = Z;

Retrouver une solution optimale : mémoriser au vol

- `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$

```
float *r[2]; int *l[2];
for (int i=0; i <= 1; i++) {
  l[i] = new int[n];   l[i][0] = i;
  r[i] = new float[n]; r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++) { // ∀ poste
  for (int i=0,k=1; i <= 1 ; i++,k=1-i) // ∀ ligne
    if (r[i][j-1] < r[k][j-1]+t[k][j-1]){ // Développement du =min
            r[i][j]=r[i][j-1]+a[i][j];} // Si on vient … de i
    else {
            r[i][j]=r[k][j-1]+t[k][j-1]+a[i][j];}//… de k
}

ropt=     min( r[0][n-1]+s[0],

               r[1][n-1]+s[1] );
```

```
X = min(Y,Z);
   ⇔
if (Y < Z)
   X = Y;
else
   X = Z;
```

Retrouver une solution optimale : mémoriser au vol

- `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$

```
float *r[2]; int *l[2];
for (int i=0; i <= 1; i++) {
  l[i] = new int[n];    l[i][0] = i;
  r[i] = new float[n]; r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++) { // ∀ poste
  for (int i=0,k=1; i <= 1 ; i++,k=1-i) // ∀ ligne
    if (r[i][j-1] < r[k][j-1]+t[k][j-1]){ // Développement du =min
      l[i][j]=i; r[i][j]=r[i][j-1]+a[i][j];} // Si on vient … de i
    else {
      l[i][j]=k; r[i][j]=r[k][j-1]+t[k][j-1]+a[i][j];}//… de k
}

ropt=     min( r[0][n-1]+s[0],

               r[1][n-1]+s[1] );
```

X = min(Y,Z);
⇔
if (Y < Z)
  X = Y;
else
  X = Z;

Retrouver une solution optimale : mémoriser au vol

- `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$

```
float *r[2]; int *l[2];
for (int i=0; i <= 1; i++) {
  l[i] = new int[n];   l[i][0] = i;
  r[i] = new float[n]; r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++) { // ∀ poste
  for (int i=0,k=1; i <= 1 ; i++,k=1-i) // ∀ ligne
    if (r[i][j-1] < r[k][j-1]+t[k][j-1]){ // Développement du =min
      l[i][j]=i; r[i][j]=r[i][j-1]+a[i][j];} // Si on vient … de i
    else {
      l[i][j]=k; r[i][j]=r[k][j-1]+t[k][j-1]+a[i][j];}//… de k
}
if (r[0][n-1]+s[0] < r[1][n-1]+s[1]) {   // Développement du =min
  lopt=0; ropt=r[0][n-1]+s[0]; }
else {
  lopt=1; ropt=r[1][n-1]+s[1]; }
```

```
X = min(Y,Z);
     ⇔
if (Y < Z)
  X = Y;
else
  X = Z;
```

Retrouver une solution optimale : restituer l'information mémorisée

- Principe
  - mémorisation partielle des solutions optimales des sous-pbs
  - `l[i][j]` : ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$
- Attention : information stockée « en partant de la fin »

```
for (int j=n-1, i=lopt; j >= 0; j--) {
   cout << "poste "<< j <<", ligne "<< i;
   i = l[i][j]; // Ligne précédente
}

// Affichage :
poste 5, ligne 0
poste 4, ligne 1
poste 3, ligne 1
...
poste 0, ligne 0
```

☛ besoin de rétablir l'ordre croissant

Rétablissement de l'ordre croissant : variante 1 : récursion et affichage « en remontant »

```
for (int j=n-1, i=lopt; j >= 0; j--) {
  cout << "poste "<< j <<", ligne "<< i;
  i = l[i][j]; // D'avant en arrière dans les lignes
}
```

```
poste 5, ligne 0
poste 4, ligne 1
poste 3, ligne 1
...
poste 0, ligne 0
```

```
printSol(lopt,n-1);
...
void printSol(i,j) {
  if (j < 0) return;
  printSol(l[i][j],j-1);                    // Descente = en arrière dans lignes
  cout << "poste "<< j <<", ligne "<< i;// Affichage à la remontée de la
}                                           //        récursion
```

```
poste 0, ligne 0
...
poste 3, ligne 1
poste 4, ligne 1
poste 5, ligne 0
```

Rétablissement de l'ordre croissant : variante 2 : inversion des « pointeurs arrières »

```
for (int j=n-1, i=lopt; j >= 0; j--) {
  cout << "poste "<< j <<", ligne "<< i;
  i = l[i][j]; // D'avant en arrière dans les lignes
}
```

```
poste 5, ligne 0
poste 4, ligne 1
poste 3, ligne 1
...
poste 0, ligne 0
```

```
int *ligne = new int[n]; // ligne[j] = ligne de destination après j, pour le poste j+1
for (int j=n-1, i=lopt; j >= 0; j--) {
  ligne[j] = i; // Mémorisation « par la fin » (indice j) du pointeur arrière (indice i)
  i = l[i][j];
}
for (int j=0; j <= n-1; j++) // Affichage par ordre croissant des postes
  cout << "poste "<< j <<", ligne "<< ligne[j];
delete [] ligne;
```

```
poste 0, ligne 0
...
poste 3, ligne 1
poste 4, ligne 1
poste 5, ligne 0
```

Retrouver une solution optimale en évitant la mémorisation

- `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$

```
float *r[2]; int *l[2];
for (int i=0; i <= 1; i++) {
  l[i] = new int[n];   l[i][0] = i;
  r[i] = new float[n]; r[i][0] = e[i]+a[i][0];
}
for (int j=1; j < n; j++) {
  for (int i=0,k=1; i <= 1 ; i++,k=1-i)
    if (r[i][j-1] < r[k][j-1]+t[k][j-1]){ // développement du min
      l[i][j]=i; r[i][j] = r[i][j-1]+a[i][j]; }
    else {
      l[i][j]=k; r[i][j] = r[k][j-1]+t[k][j-1]+a[i][j]; }
}
if (r[0][n-1]+s[0] < r[1][n-1]+s[1]) {
  lopt=0; ropt=r[0][n-1]+s[0]; }
else { // développement du min
  lopt=1; ropt=r[1][n-1]+s[1]; }
```

x = **min**(y,z);
⇔
if (y < z)
  x = y;
else
  x = z;

Observation : la valeur de `l[i][j]` peut être connue a posteriori en testant la valeur de `r[i][j]` par rapport à `r[i][j-1]+a[i][j]`.

Retrouver une solution optimale : reconstruction a posteriori

- `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$

```
/* Calcul de r et ropt sans mémorisation de l[i][j], comme précédemment */
...

/* Calcul de l et lopt a posteriori */
int *l[2];
for (int i=0; i <= 1; i++)
{
  l[i] = new int[n];
  l[i][0] = i;
}
for (int j=1; j < n; j++)
  for (int i=0,k=1; i <= 1 ; i++,k=1-i)
    l[i][j] = (r[i][j] == r[i][j-1]+a[i][j]) ? i : k;
lopt = (r[0][n-1]+s[0] < r[1][n-1]+s[1]) ? 0 : 1;
```

$$\boxed{r_{i,j} = \min(r_{i,j-1}\,,\, r_{k,j-1} + t_{k,j-1}) + a_{i,j}}$$

Résumé

- Éviter une complexité exponentielle
  - choisir un ordre d'examen de sous-problèmes
  - mémoriser les meilleurs sous-résultats (partiel = suffisant)
- Solution(s) récursive(s) vs itérative(s)
  - récursive ⇒ simple (proche maths), un peu moins efficace
  - **itérative** ⇒ plus de travail mais **complexité sue/maîtrisée**
  - compromis stockage / recalculs
- Exhiber la meilleure solution
  - mémorisation ou non des choix ⇒ tests ultérieurs sinon
  - chemin connu depuis la fin ⇒ travail d'inversion

Exemple 2 : multiplication de matrices

- Soient deux matrices $\mathrm{A} : p \times q$ et $\mathrm{B} : q \times r$
- On calcule $\mathrm{C} = \mathrm{AB} : p \times r$

```
for (int i=1; i <= p; i++)
  for (int j=1; j <= r; j++) {
    C[i][j] = 0;
    for (int k=1; k <= q; k++)
      C[i][j] += A[i][k] * B[k][j]; }
```

$$\begin{pmatrix} C_{1,1} & \cdots & C_{1,r} \\ \vdots & \ddots & \vdots \\ C_{p,1} & \cdots & C_{p,r} \end{pmatrix} = \begin{pmatrix} A_{1,1} & \cdots & A_{1,q} \\ \vdots & \ddots & \vdots \\ A_{p,1} & \cdots & A_{p,q} \end{pmatrix} \begin{pmatrix} B_{1,1} & \cdots & B_{1,r} \\ \vdots & \ddots & \vdots \\ B_{q,1} & \cdots & B_{q,r} \end{pmatrix}$$

Exemple 2 : multiplication de matrices

- Soient $\mathrm{A} : p \times q$ et $\mathrm{B} : q \times r$, on calcule $\mathrm{C} = \mathrm{AB} : p \times r$

```
for (int i=1; i <= p; i++)
  for (int j=1; j <= r; j++) {
    C[i][j] = 0;
    for (int k=1; k <= q; k++)
      C[i][j] += A[i][k] * B[k][j]; }
```

- Complexité ?
  - mesurée par rapport à quoi ?

Exemple 2 : multiplication de matrices

- Soient $\mathrm{A} : p \times q$ et $\mathrm{B} : q \times r$, on calcule $\mathrm{C} = \mathrm{AB} : p \times r$

```
for (int i=1; i <= p; i++)
  for (int j=1; j <= r; j++) {
    C[i][j] = 0;
    for (int k=1; k <= q; k++)
      C[i][j] += A[i][k] * B[k][j]; }
```

- Complexité
  - C calculé avec pqr multiplications : $\Theta(n^3)$ pour matrice $\mathrm{n}\times\mathrm{n}$
  - N.B. Il existe de meilleurs algorithmes théoriques
    - Strassen [1969] : $O(n^{2,8074})$ a ouvert nouveau champ de recherche
    - Coppersmith-Winograd [1987] : $O(n^{2,3755})$ avec constante énorme
    - Stothers [2010], Williams [2011], Le Gall [2014] : $O(n^{2,3729})$ idem
    - problèmes possibles de stabilité numérique

Exemple 2 : multiplication de matrices

- Associativité de la multiplication de matrices
  - $(A\,B)\,C = A\,(B\,C)$

- Importance pratique de l'ordre
- Exemple
  - soient $A : 10 \times 100,\ B : 100 \times 5,\ C : 5 \times 50$
  - $(A\,B)\,C : (10 \cdot 100 \cdot 5) + (10 \cdot 5 \cdot 50) = \mathbf{7.500}$ multiplications
  - $A\,(B\,C) : (100 \cdot 5 \cdot 50) + (10 \cdot 100 \cdot 50) = \mathbf{75.000}$ multiplications

  intuition : $(B\,C)$ augmente le nb de coefficients

Exemple 2 : multiplication de matrices

- Problème (matrix-chain multiplication)
  - soient n matrices $(A_i)_{1 \leq i \leq n}$ de dimension $(p_{i\text{-}1} \times p_i)_{1 \leq i \leq n}$
  - parenthéser le produit $A_1 \ldots A_n$ pour réduire au plus le nombre total de multiplications de scalaires

- N.B. C'est le calcul de la **meilleure position des parenthèses**, pas le calcul de la matrice produit !

- Notation : $A_{i..j} = A_i \ldots A_j$ pour $i \leq j$

Nombre de parenthésages

- Soient n matrices $(A_i)_{1 \leq i \leq n}$ de dimension $(p_{i\text{-}1} \times p_i)_{1 \leq i \leq n}$
- Combien de parenthésages du produit $A_1 \ldots A_n$?

Nombre de parenthésages

- Soient n matrices $(A_i)_{1 \le i \le n}$ de dimension $(p_{i\text{-}1} \times p_i)_{1 \le i \le n}$
- Combien de parenthésages du produit $A_1 \dots A_n$?
  - $P(n)$: nb de parenthésages d'un produit de n matrices
  - $$P(n) = \left\{ \begin{matrix} 1 & \text{si } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \ge 2 \end{matrix} \right\} = \frac{(2(n-1))!}{(n-1)!\,n!}$$

    = nombre d'arbres binaires à n feuilles

    → nombres de Catalan, qui croissent en $\Omega(4^n/n^{3/2})$

☛ On ne peut pas les énumérer en pratique

Structure d'un parenthésage optimal

- Un parenthésage de $A_{i..j}$ (pour $i < j$) doit séparer le produit entre A_k et A_{k+1} pour $i \leq k < j$, avec les calculs
  - $M_1 = A_{i..k}$
  - $M_2 = A_{k+1..j}$
  - $M_3 = M_1 \times M_2$
- Si un parenthésage optimal de $A_{i..j}$ sépare le produit entre A_k et A_{k+1} alors les parenthésages sous-jacents de $A_{i..k}$ et $A_{k+1..j}$ sont optimaux [preuve par l'absurde]
  - ☛ **propriété de sous-structure optimale**

Construction d'un parenthésage optimal

- Idée générale
  - on coupe le pb pour $A_{i..j}$ en 2 sous-pbs $A_{i..k}$, $A_{k+1..j}$ $\forall i \leq k < j$
  - on trouve une solution optimale pour $A_{i..k}$
  - on trouve une solution optimale pour $A_{k+1..j}$
  - on les combine pour former une solution optimale pour $A_{i..j}$
    - tailles (et valeurs) de $A_{i..k}$ et $A_{k+1..j}$ indépendants de leur parenthésage :
      - $A_{i..k}$: matrice $p_{i-1} \times p_k$
      - $A_{k+1..j}$: matrice $p_k \times p_j$
    - nombre de multiplications scalaires pour $A_{i..k} \times A_{k+1..j}$: $p_{i-1}\, p_k\, p_j$

Solution récursive

- $m_{i,j}$: nombre minimum de multiplications scalaires pour calculer les coefficients de $A_{i..j}$
  - $A_{i..j} = (A_i \dots A_k)\,(A_{k+1} \dots A_j)$ pour un certain $k \in \{i, \dots, j-1\}$
  - $m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \le k < j}\left(m_{i,k} + m_{k+1,j} + p_{i-1}\, p_k\, p_j\right) & \text{si } i < j \end{cases}$

Équation de Bellman

- Complexité d'une implémentation directe (récursive) ?

Solution récursive

- $m_{i,j}$: nombre minimum de multiplications scalaires pour calculer les coefficients de $A_{i..j}$
  - $A_{i..j} = (A_i \dots A_k)\,(A_{k+1} \dots A_j)$ pour un certain $k \in \{i, \dots, j-1\}$
  - $m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \le k < j} \left(m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j \right) & \text{si } i < j \end{cases}$ Équation de Bellman
- Complexité d'une implémentation directe (récursive) : exponentielle !

Solution récursive

- $m_{i,j}$: nombre minimum de multiplications scalaires pour calculer les coefficients de $A_{i..j}$
  - $A_{i..j} = (A_i \ldots A_k)\,(A_{k+1} \ldots A_j)$ pour un certain $k \in \{i, \ldots, j-1\}$
  - $m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \left(m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j \right) & \text{si } i < j \end{cases}$ Équation de Bellman
- Combien de sous-pbs à examiner ? (combien de $m_{i,j}$?)

Solution récursive

- $m_{i,j}$: nombre minimum de multiplications scalaires pour calculer les coefficients de $A_{i..j}$
  - $A_{i..j} = (A_i \dots A_k)\,(A_{k+1} \dots A_j)$ pour un certain $k \in \{i, \dots, j-1\}$
  - $m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \le k < j} \left(m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j \right) & \text{si } i < j \end{cases}$

    Équation de Bellman
- Combien de sous-pbs à examiner ? (combien de $m_{i,j}$?)
  - $\sum_{j=1}^{n} (1 + j) = n + \frac{n(n+1)}{2} = \Theta(n^2)$

    Différence avec la ligne d'assemblage : ici min sur un nombre variable de valeurs
  - bonne propriété de recouvrement des sous-problèmes : on se repose les mêmes questions à différents niveaux

Solution itérative

- Calculer $m_{i,j}$ pour $1 \leq i \leq j \leq n$

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \left(m_{i,k} + m_{k+1,j} + p_{i-1}\, p_k\, p_j \right) & \text{si } i < j \end{cases}$$

- Résoudre les problèmes « dans l'ordre » : comment ?

Solution itérative

- Calculer $m_{i,j}$ pour $1 \leq i \leq j \leq n$

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \left(m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j \right) & \text{si } i < j \end{cases}$$

- Résoudre les problèmes « dans l'ordre »
- Procéder par ordre croissant de **nombres de matrices dans $\boldsymbol{A_{i..j}}$**, c.-à-d. par ordre croissant de $l = j - i + 1$
  - sous-problèmes
    - nombre de matrices dans $A_{i..k} = k - i + 1 \;\; < j - i + 1 = l$
    - nombre de matrices dans $A_{k+1..j} = j - k \quad < j - i + 1 = l$

Solution itérative

$$l = j - i + 1$$

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j}\left(m_{i,k} + m_{k+1,j} + p_{i-1}\, p_k\, p_j\right) & \text{si } i < j \end{cases}$$

```
for (i=1; i <= n; i++) // Cas l = 1
  m[i][i] = 0;
for (l=2; l <= n; l++) // Cas l ≥ 2
  for (i=1,j=i+l-1; j <= n; i++,j++){
    m[i][j] = MAXINT;
    for (k=i; k <= j-1; k++) {
      u = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
      if (u < m[i][j]) { // Calcul du min
        m[i][j] = u;
        parenth[i][j] = k; // Mémoriser la meilleure parenth.
   }  }  }
```

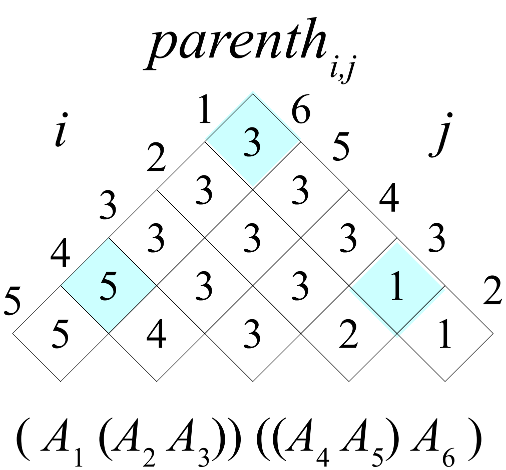
```
t = new int[n];
x = min(t);
      ⇔
x = MAXINT;
for (i=0; i<n ; i++)
  if (t[i] < x)
    x = t;
```

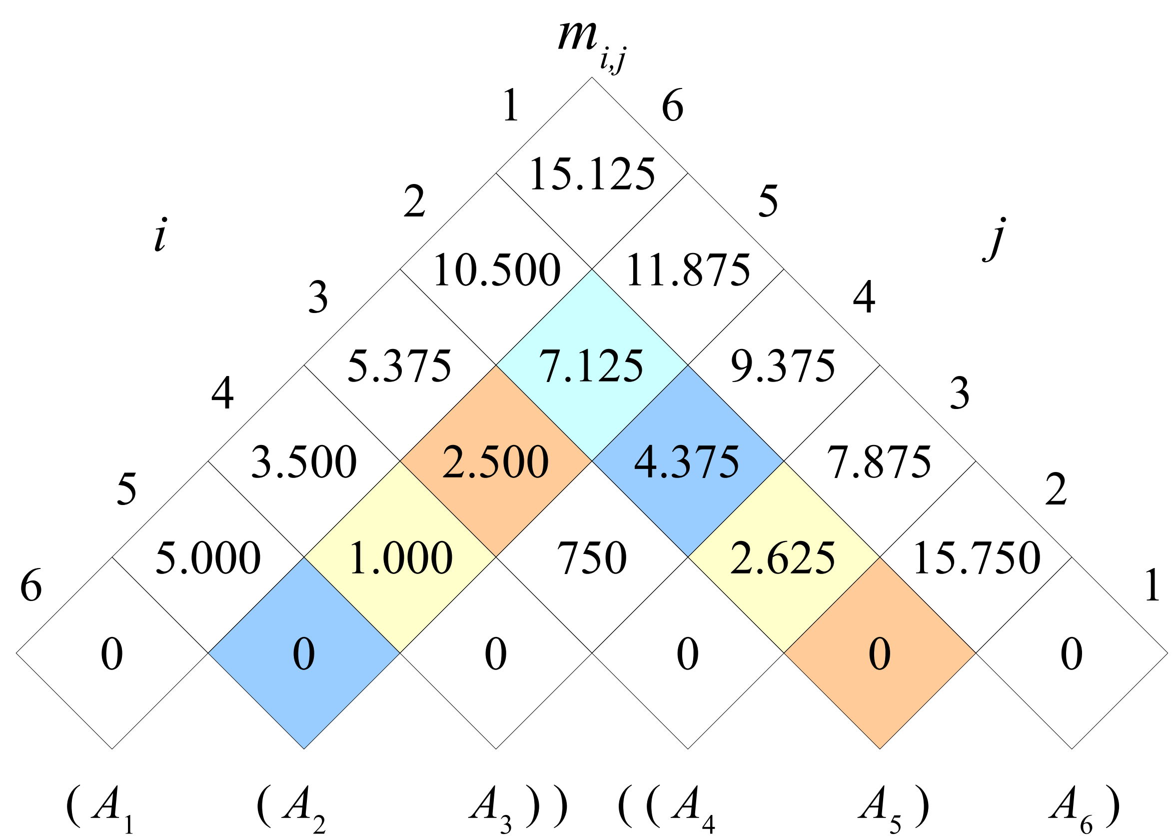
Exemple de calcul de parenthésage

Matrice	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

p_0	30
p_1	35
p_2	15
p_3	5
p_4	10
p_5	20
p_6	25

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \le k < j}\left(m_{i,k} + m_{k+1,j} + p_{i-1}\, p_k\, p_j\right) & \text{si } i < j \end{cases}$$





Solution itérative

```
for (i=1; i <= n; i++)
  m[i][i] = 0;
for (l=2; l <= n; l++)
  for (i=1,j=i+l-1; j <= n; i++,j++) {
    m[i][j] = MAXINT;
    for (k=i; k <= j-1; k++) {
      u = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
      if (u < m[i][j]) {
        m[i][j] = u;
        parenth[i][j] = k; // mémoris. du pt de parenth.
  } } }
```

- Complexité :
  - en temps = ?   , en espace = ?

Solution itérative

```
for (i=1; i <= n; i++)
  m[i][i] = 0;
for (l=2; l <= n; l++)
  for (i=1,j=i+l-1; j <= n; i++,j++) {
    m[i][j] = MAXINT;
    for (k=i; k <= j-1; k++) {
      u = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
      if (u < m[i][j]) {
        m[i][j] = u;
        parenth[i][j] = k; // mémor. du pt de parenth.
  } } }
```

- Complexité :
  - en temps = $\Theta(n^3)$, en espace = $\Theta(n^2)$

Conditions générales d'application des principes de la programmation dynamique

- **Sous-structure optimale (propriété à vérifier!)**
  - une solution optimale du problème contient des solutions optimales de sous-problèmes
  - les solutions optimales de sous-problèmes permettent de construire un solution optimale du problème
- **Superposition des sous-problèmes (à vérifier !)**
  - les mêmes sous-pbs doivent avoir à être résolus plusieurs fois (et même de nombreuses fois...) ⇒ les sous-problèmes rencontrés ne doivent pas être tous différents

☛ Procédé « bottom-up » (de bas en haut)

Comment résoudre un pb par prog. dyn. ? **(si c'est applicable)**

1. Supposer qu'on dispose d'une solution optimale
2. Montrer qu'elle est le résultat d'un choix, qui suppose de résoudre un ensemble de sous-problèmes
3. Montrer que les solutions des sous-problèmes dans une solution optimale sont elles-mêmes optimales
   - ☛ par l'absurde : ∀ sous-problème de la solution optimale, supposer qu'une solution du sous-pb n'est pas optimale et construire une solution encore meilleure (→ contrad.)
4. Écrire les équations de récurrence entre pb et sous-pbs
5. Les implémenter efficacement (itér. ou récur.+mémoïs.)

Comment résoudre un pb par prog. dyn. ? **(si c'est applicable)**

1. Supposer qu'on dispose d'une solution optimale
2. Montrer qu'elle est le résultat d'un choix, qui suppose de résoudre un ensemble de sous-problèmes
3. Montrer que les solutions des sous-problèmes dans une solution optimale sont elles-mêmes optimales
   - ☛ par l'absurde : ∀ sous-problème de la solution optimale, supposer qu'une solution du sous-pb n'est pas optimale et construire une solution encore meilleure (→ contrad.)

Indispensable !

4. Écrire les équations de récurrence entre pb et sous-pbs
5. Les implémenter efficacement (itér. ou récur.+mémoïs.)

Comment choisir les sous-problèmes ?

1. Essayer des sous-pbs simples (1 var.) : $\mathrm{pb}(j)=\mathrm{f}(\mathrm{pb}(j\text{-}1))$
   - ex. temps minimum jusqu'aux postes $P_{1,\,j}$ et $P_{2,\,j}$ en fonction des temps mininum jusqu'à $P_{1,\,j\text{-}1}$ et $P_{2,\,j\text{-}1}$
   - ex. meilleur parenthésage du produit $A_{1..j}$ en fonction du meilleur parenthésage du produit $A_{1..j\text{-}1}$
2. Généraliser si ça ne marche pas (2 variables ou plus) : $\mathrm{pb}(i,j) = \mathrm{f}(\mathrm{pb}(i',j')$ pour des $i'<i, j'<j)$
   - ex. meilleur parenthésage du produit $A_{i..j}$ en fonction des meilleurs pour les produits $A_{i..k}$ et $A_{k+1..j}$, pour $i < k \leq j$
3. Pour finir, instancier sur le problème initial, de taille n
   - ex. $j=n$, ou bien $i=1$ et $j=n$, etc.

Complexité d'une solution par programmation dynamique (1)

- Ce qui varie :
  - nb de sous-problèmes utilisés dans une solution optimale
  - nb de choix pour déterminer quels sous-problèmes utiliser dans une solution optimale
- Ex. ligne d'assemblage
  - $2n$ sous-problèmes : $P_{i,j}$ pour $i \in \{1,2\}$, $1 \leq j \leq n$
  - 2 choix : $P_{i,j} \rightarrow P_{i,j-1}$ ou $P_{k,j-1}$
- Ex. produit matriciel
  - $\Theta(n^2)$ sous-problèmes : $A_{i..j}$ pour $1 \leq i < j \leq n$
  - $O(n)$ choix : $A_{i..j} \rightarrow A_{i..k} A_{k+1..j}$ pour $i < k \leq j$

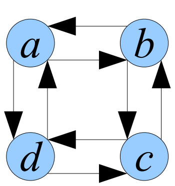
Complexité d'une solution par programmation dynamique (2)

- Complexité
  - coût solution = coût des sous-problèmes $\otimes$ coût des choix
  - complexité $\approx$ nb sous-pbs $\times$ nb choix par sous-pbs (en général)
- Ex. ligne d'assemblage : $\Theta(n)$
  - $2n$ sous-problèmes : $P_{i,j}$ pour $i \in \{1,2\},\ 1 \leq j \leq n$
  - 2 choix : $P_{i,j} \rightarrow P_{i,j-1}$ ou $P_{k,j-1}$
- Ex. produit matriciel : $\Theta(n^3)$
  - $\Theta(n^2)$ sous-problèmes : $A_{i..j}$ pour $1 \leq i < j \leq n$
  - $O(n)$ choix : $A_{i..j} \rightarrow A_{i..k} A_{k+1..j}$ pour $i < k \leq j$

Attention

- Plus court chemin (sans cycle) dans graphe orienté
  - si $u \xrightarrow{p} v$ minimal, décomposé en $u \xrightarrow{q} w \xrightarrow{r} v$ alors $u \xrightarrow{q} w$ et $w \xrightarrow{r} v$ minimaux [par l'absurde]

    ☛ sous-structure optimale
- Plus long chemin (sans cycle) dans graphe orienté
  - si $u \xrightarrow{p} v$ maximal, décomposé en $u \xrightarrow{q} w \xrightarrow{r} v$ alors $u \xrightarrow{q} w$ et $w \xrightarrow{r} v$ pas forcément maximaux
    - ex. $a \to b \to c$ maximal n'implique pas $a \to b$ maximal



  - pb : combinaison 2 chemins maximaux $\neq$ chemin maximal
    - ex. $a \to b \to c \to d$ et $d \to a \to b \to c$

    ☛ pas de sous-structure optimale (non-indépendance des sous-pbs)

    ☛ principes de la programmation dynamique pas applicables

Implémentation

- Solution récursive
  - écriture directe : équations mathématiques + mémoïsation
  - ne dispense pas de l'étude des sous-pbs pour vérifier que les conditions d'applications de la prog. dyn. sont remplies
  - risque : complexité exponentielle sans qu'on le sache
- Solution itérative
  - un peu plus de travail
    - stockage réfléchi de solutions optimales de sous-problèmes
    - pas d'examen répété de sous-problèmes
  - mais complexité maîtrisée
    - en temps et en espace (allocation et libération mémoire)

Reconstruction d'une solution optimale

- Mémorisation explicite d'information
  - ex. `l[i][j]` : n° ligne de provenance du plus rapide chemin arrivant en $P_{i,j}$
  - ex. `parenth[i][j]` : position du produit dans $A_{i..j}$
- Solution optimale parfois identifiable à travers les quantités optimales stockées
  - ex. `l[i][j]=i` $\Leftrightarrow$ `r[i][j] == r[i][j-1]+a[i][j]`
    - choix retrouvé en $O(1)$: économise du stockage contre du temps
  - contrex. : retrouver le point k du meilleur parenthésage
    - nécessite d'examiner $j-i$ possibilités : coûteux en temps

Quelques autres exemples d'application de la programmation dynamique

- Découpage d'un texte de n mots en lignes pour minimiser les espaces vides sur chaque ligne (LaTeX, pas Word)
- Plus longue sous-séquence croissante
- Algorithme de Viterbi

  Étant donnée une séquence d'événements observés, produits par un Modèle de Markov Caché (HMM ≈ système de transition à états finis avec probabilités), trouver la séquence d'états (cachés) la plus probable
- Plus court chemin dans un graphe pondéré
- Pb du sac à dos 0-1 : n objets de poids $(w_i)_{1 \leq i \leq n}$ et valeurs $(v_i)_{1 \leq i \leq n}$ entiers, à choisir et mettre au mieux (valeur totale) dans un sac de poids maximum W
- Analyse syntaxique pour grammaire hors contexte (algo. CYK)
- Arbre binaire de recherche optimal selon la fréquence d'occurrence
- ...

Plus longue sous-séquence commune

- Comparaison d'ADN
  - recherche de similarité :
    - ex. sous-chaîne identique : apparition des bases dans même ordre
  - exemple
    - S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
    - S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA
    - plus longue sous-séq commune : GTCGTCGGAAGCCGGCCGAA
- Recherche par programmation dynamique
  - $O(mn)$ pour deux mots de taille m et n
  - en fait pas praticable pour grandes chaînes, OK pour petites

Pourquoi « programmation dynamique » ?

- Richard Bellman (USA), années 1940-1955 :
  - « **dynamic programming** »

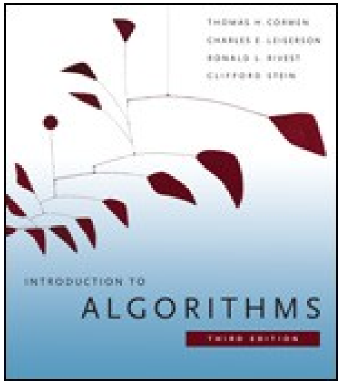
- « **programmation** » au sens d'ordonnancement des problèmes, de planification
- organisation dans le temps → « **dynamique** »

  [même sens de « programmation » dans « programmation linéaire » (optimisation)]

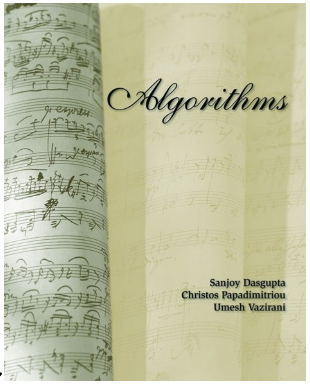
- À l'époque, ministre de la défense hostile aux recherches en maths → choix d'un nom « positif » pour éviter la confrontation
  - impossible d'utiliser dynamique dans un sens péjoratif
  - quelque chose que « même un membre du congrès ne pourrait désapprouver »

Bibliographie

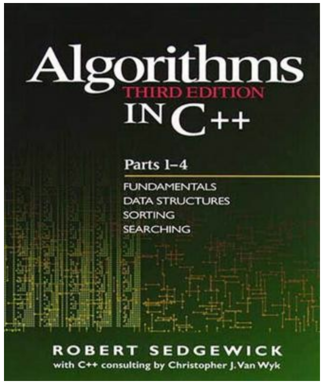
- *Introduction to Algorithms.* T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein. The MIT Press. 3rd edition, 2009.

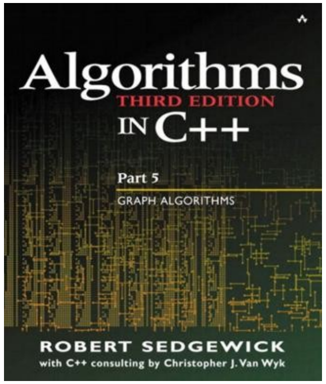


- *Algorithms.* S. Dasgupta, C. Papadimitriou, U. Vazirani. McGraw-Hill. 1st edition, 2006.



- *Algorithms in C++.* R. Sedgewick. Addison-Wesley, 3rd edition, 2002.





Distance de Levenshtein (1965) = distance d'édition

- Nombre minimum de modifications pour passer d'une chaîne à une autre :
  - suppression d'un caractère
  - insertion d'un caractère
  - remplacement d'un caractère par un autre
    - ex. d_L(ponts,hotes) = 3
- Mesure de la similarité de deux chaînes de caractères
  - applications : vérificateur orthographique, OCR...
  - généralisation du pb de plus longue séquence commune
- Complexité : $O(mn)$ pour deux mots de taille m et n

Distance de Damerau-Levenshtein

- Nombre minimum de modifications pour passer d'une chaîne à une autre :
  - suppression d'un caractère
  - insertion d'un caractère
  - remplacement d'un caractère par un autre
  - transposition de deux caractères successifs
    - ex. d_{DL}(écoles,éclose) = 2
      d_L(écoles,éclose) = 3

écoles, o ↔ l =
écloes, s ↔ e =
éclose

écoles, +l =
écloles, l → s =
écloses, -s =
éclose

- Couvrirait ≈80% des fautes d'orthographe (anglais)
- Complexité : $O(mn)$ pour deux mots de taille m et n

TP : distance d'édition

1. **Fermez votre ordinateur !**
2. Étudiez le calcul par programmation dynamique de la distance de Levenshtein $d_L(s,s')$ entre deux chaînes s et s'
   - **suivez la méthodologie du cours !**
   - **indice :** on admet que les sous-problèmes sont les distances entre préfixes
   - **rédigez** la preuve de la sous-structure optimale et la formule de Bellman (qq lignes)

chaîne $s = c_1 \ldots c_n$
préfixe $s_i = c_1 \ldots c_i$ $1 \leq i \leq n$ ou $s_0 = \emptyset$

3. Implémentez des solutions récursives, sans et avec mémoïsation [≈ **10 LOC !]**
   Utiliser un seul cœur de fonction prenant un pointeur sur le cache (pointeur nul pour ne pas mémoïser).
4. Quelles sont leur complexité en espace & en temps (approx : polyn. ou expon.) ?
5. Comparez et discutez leur temps d'exécution
6. Implémentez un algorithme itératif [≈ **10 LOC !]**
7. Quelle est sa complexité en espace et en temps ?
8. Comparez et discutez son temps d'exécution

```
// Temps d'exécution
#include <ctime>
clock_t t1 = clock();
traitement();
clock_t t2 = clock();
cout << (t2-t1) /(float)
  CLOCKS_PER_SEC ;
```

TP : distance d'édition

9. Affichez la suite de modifs élémentaires pour aller de s à s'.
   La fonction prend en paramètre le tableau des distances entre préfixes, retourné par la fonction de la question 6.
   - Chemin Levenshtein : suppression du *o*, substitution de *e* en *o*, ajout de *e*.
   L'affichage est sous la forme :
   ```
   ecoles
   ec-o-les : ecles
   ecle->os : eclos
   eclos+e+
   ```
10. Étendez ce travail à la distance de Damerau-Levenshtein (ajoutez un paramètre `bool` aux fonctions implémentées)
    - Affichage chemin Damerau Levenshtein : transposition *o* et *l*, puis *e* et *s*
    ```
    ecoles
    ec(ol)es : ecloes
    eclo(es)
    ```
11. Proposez une version itérative linéaire <u>en espace</u>