M1 Info - Optimisation et Recherche Opérationnelle

Cours 4 - Programmation dynamique

Le voyageur de commerce

Semestre Automne 2020-2021 - Université Claude Bernard Lyon 1

Christophe Crespelle
christophe.crespelle@inria.fr

département
Informatique

Faculté des Sciences et Technologies
Université Claude Bernard Lyon 1

Problème du voyageur de commerce

- **Entrée :** un ensemble de villes $\{c_1, c_2, \ldots, c_n\}$ et une fonction de distance $d(c_i, c_j)$ definie pour tous les couples (c_i, c_j).

Problème du voyageur de commerce

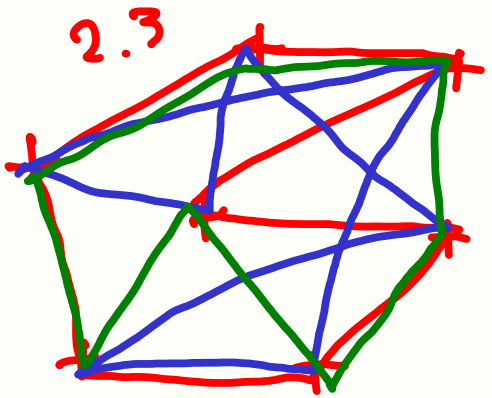
- **Entrée :** un ensemble de villes $\{c_1, c_2, \ldots, c_n\}$ et une fonction de distance $d(c_i, c_j)$ definie pour tous les couples (c_i, c_j).

Définition (Tour et tour minimum)

Un tour est une permutation $\pi = (c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)})$ des villes.
La longueur d'un tour π est

$$\sum_{i \in [\![1, n-1]\!]} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

Un tour minimum est un tour dont la longueur est minimum.



Tour 1
Tour 2
Tour 3

longueur du tour
=
somme des distances entre 2 sommets consécutifs dans le tour (en revenant au point de départ)

Problème du voyageur de commerce

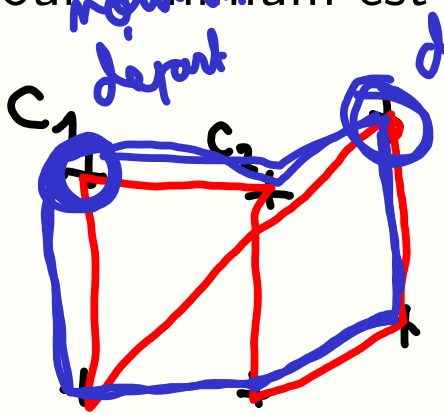
- **Entrée :** un ensemble de villes $\{c_1, c_2, \ldots, c_n\}$ et une fonction de distance $d(c_i, c_j)$ definie pour tous les couples (c_i, c_j).

Définition (Tour et tour minimum)

Un tour est une permutation $\pi = (c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)})$ des villes.
La longueur d'un tour π est

$$\sum_{i \in [\![1, n-1]\!]} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

Un tour minimum est un tour dont la longueur est minimum.



Problème du voyageur de commerce

- **Entrée :** un ensemble de villes $\{c_1, c_2, \ldots, c_n\}$ et une fonction de distance $d(c_i, c_j)$ definie pour tous les couples (c_i, c_j).

Définition (Tour et tour minimum)

Un tour est une permutation $\pi = (c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)})$ des villes.
La longueur d'un tour π est

$$\sum_{i \in [\![1, n-1]\!]} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

Un tour minimum est un tour dont la longueur est minimum.

- **Sortie :** un tour minimum de $\{c_1, c_2, \ldots, c_n\}$.

Problème du voyageur de commerce

Applications classiques :

- en logistique : optimisation des transports

Problème du voyageur de commerce

Applications classiques :

- en logistique : optimisation des transports
- en electronique : minimisation des cablages entre composants des circuits integres

Problème du voyageur de commerce

Applications classiques :

- en logistique : optimisation des transports
- en electronique : minimisation des cablages entre composants des circuits integres

Difficulte de calcul : **NP-complet**

Problème du voyageur de commerce

Applications classiques :

- en logistique : optimisation des transports
- en electronique : minimisation des cablages entre composants des circuits integres

Difficulte de calcul : **NP-complet**

On va faire un algorithme exponentiel ($O^*(2^n)$) pour le resoudre, par la programmation dynamique.

Approche brute force

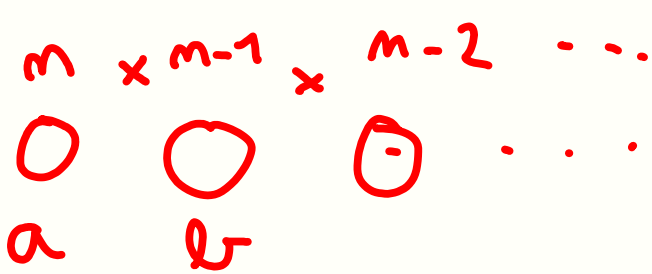
Algo brute force :

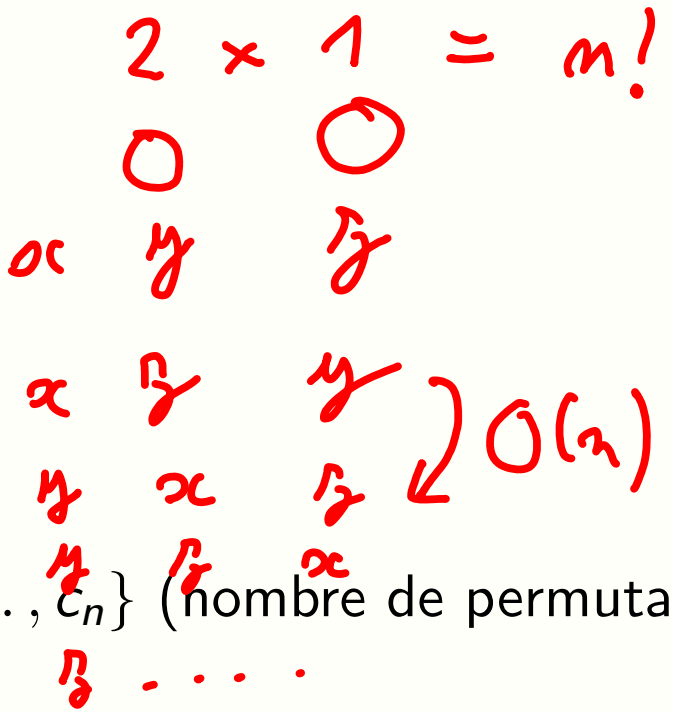
- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

Approche brute force

Algo brute force :

- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

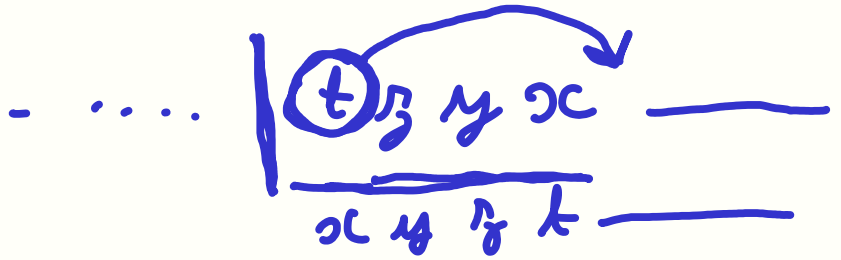




Complexite :

- il y a $n!$ tour π de $\{c_1, c_2, \ldots, c_n\}$ (nombre de permutations sur n elements)

Approche brute force



Algo brute force :

- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

une permutation : $a_n a_{n-1} \quad a_h\ a_{h-1} \cdots a_{i+1} a_i \ldots a_1$

$n!$

Complexite :

- il y a $n!$ tour π de $\{c_1, c_2, \ldots, c_n\}$ (nombre de permutations sur n elements)
- calculer la longueur de π prend $O(n)$

Total : $O(n \cdot n!)$

Approche brute force

Algo brute force :

- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

Complexite :

- il y a $n!$ tour π de $\{c_1, c_2, \ldots, c_n\}$ (nombre de permutations sur n elements)
- calculer la longueur de π prend $O(n)$

Total : $O(n \cdot n!)$

On va faire un algo en 2^n par la programmation dynamique.

Gain de complexite

Note importante : taille de $n!$ comparee a 2^n ?

Gain de complexite

Note importante : taille de $n!$ comparee a 2^n ?

Maths (Stirling) $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$

Retenez $n! = n^n$: enorme, bien plus gros que 2^n

Gain de complexite

Note importante : taille de $n!$ comparee a 2^n ?

Maths (Stirling) $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$

Retenez $n! = n^n$: enorme, bien plus gros que 2^n

Conclusion : complexite en $n!$ est redhibitoire

On va faire un algo exponentiel, en 2^n, par la programmation dynamique.

Programmation dynamique Richard Bellman

Idee generale : stocker dans une table tous les resultats intermediaires

2 façon de le voir

l'été

remplissage de table

fondamental

espace ⟷ temps

plus de gagner

Programmation dynamique

Idee generale : stocker dans une table tous les resultats intermediaires

Conditions de mise en oeuvre : il faut une "formule de reccurence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

Programmation dynamique

Idee generale : stocker dans une table tous les resultats intermediaires

Conditions de mise en oeuvre : il faut une "formule de reccurence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)

Programmation dynamique

Idee generale : stocker dans une table tous les resultats intermediaires

Conditions de mise en oeuvre : il faut une "formule de reccurence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)
- et surtout, sans stocker trop de resultats intermediaires

Programmation dynamique

Idee generale : stocker dans une table tous les resultats intermediaires

Conditions de mise en oeuvre : il faut une "formule de reccurence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)
- et surtout, sans stocker trop de resultats intermediaires

gain r/t a brute force : comme on stocke, on ne perd pas de temps a recalculer les resultats intermediaires

Programmation dynamique

Idee generale : stocker dans une table tous les resultats intermediaires

Conditions de mise en oeuvre : il faut une "formule de reccurence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)
- et surtout, sans stocker trop de resultats intermediaires

gain r/t a brute force : comme on stocke, on ne perd pas de temps a recalculer les resultats intermediaires

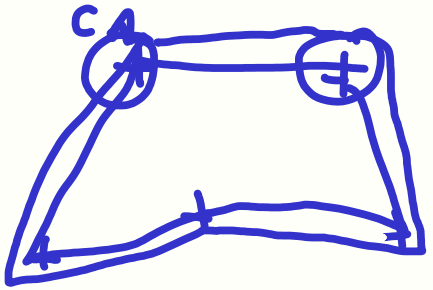
contrepartie : ca prend beaucoup d'espace (en fait on echange de l'espace contre du temps de calcul)

Programmation dynamique pour le voyageur de commerce

Remarque

On peut toujours commencer le tour sur la ville de notre choix : on choisit c_1.

point de départ fixé toujours



Programmation dynamique pour le voyageur de commerce

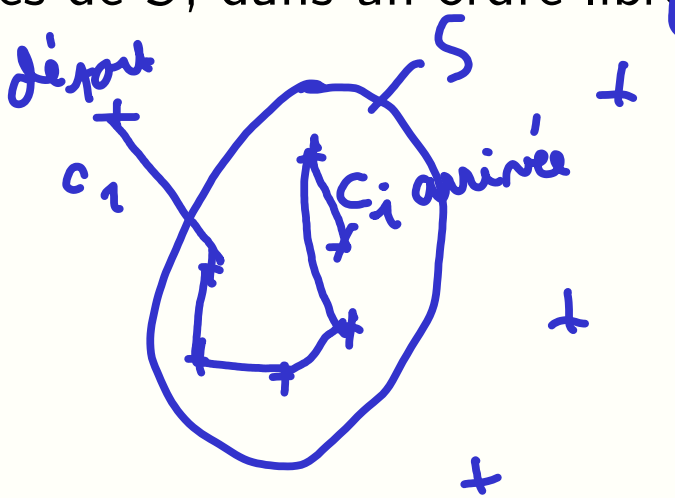
Remarque

On peut toujours commencer le tour sur la ville de notre choix : on choisit c_1.

Définition

Pour $S \subseteq \{c_2, \ldots, c_n\}$ et $c_i \in S$, on note $OPT[S, c_i]$ la longueur minimum d'un parcours qui :

- commence en c_1
- parcours les villes de S, dans un ordre libre
- finit en c_i



Programmation dynamique pour le voyageur de commerce

La formule de recurrence :

- si $|S| = 1$, c.a.d. $S = \{c_i\}$, $i \neq 1$, on a $OPT[S, c_i] = d(c_1, c_j)$.

Programmation dynamique pour le voyageur de commerce

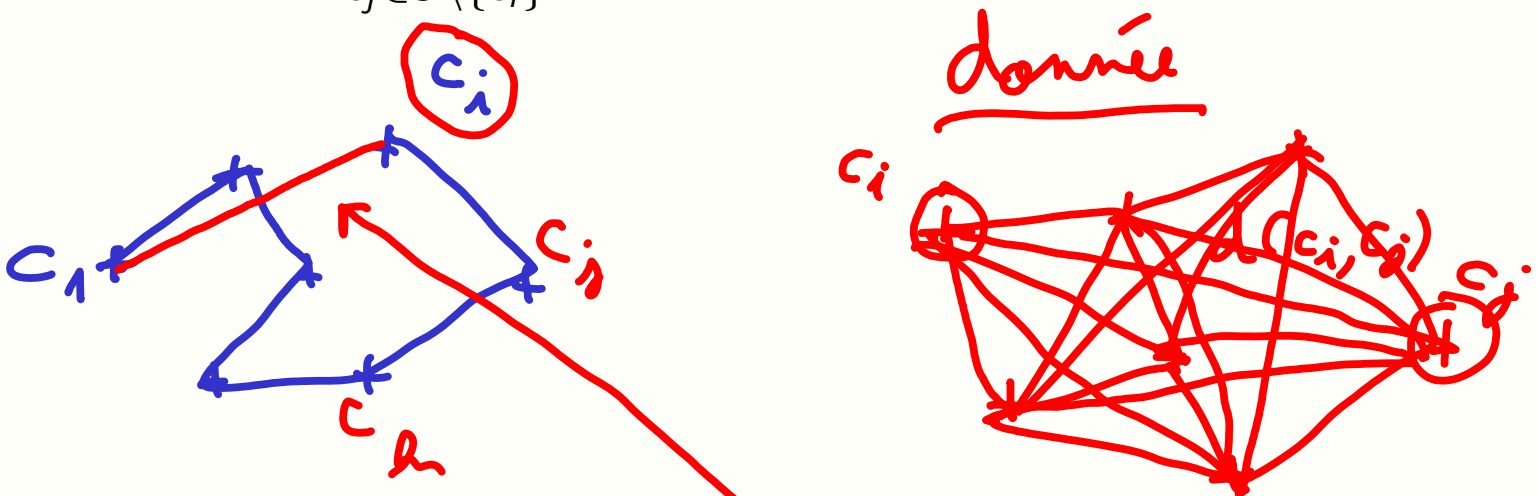
La formule de recurrence :

- si $|S| = 1$, c.a.d. $S = \{c_i\}$, $i \neq 1$, on a $OPT[S, c_i] = d(c_1, c_j)$.
- si $|S| > 1$, alors
  $OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$

Programmation dynamique pour le voyageur de commerce

La formule de recurrence :

- si $|S| = 1$, c.a.d. $S = \{c_i\}$, $i \neq 1$, on a $OPT[S, c_i] = d(c_1, c_j)$.
- si $|S| > 1$, alors
  $OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$



La reponse au probleme : la longueur d'un tour minimum est
$OPT = \min_{i \in [\![2,n]\!]} \{OPT[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1)\}$.

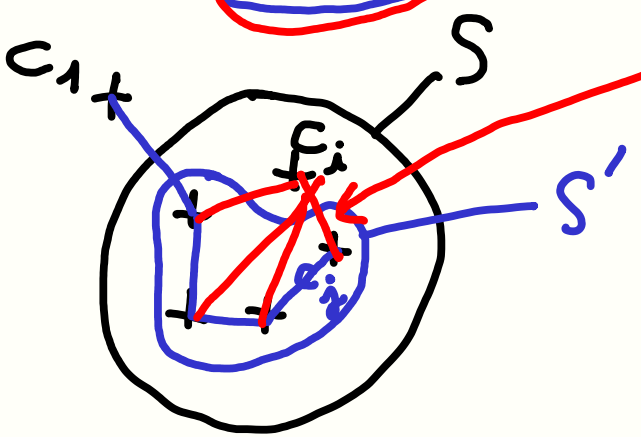
Programmation dynamique pour le voyageur de commerce

La formule de recurrence :

- si $|S| = 1$, c.a.d. $S = \{c_i\}$, $i \neq 1$, on a $OPT[S, c_i] = d(c_1, c_i)$.
- si $|S| > 1$, alors

$$OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$$

Formule de récurrence



$|S'| = |S| - 1$

La reponse au probleme : la longueur d'un tour minimum est

$$OPT = \min_{i \in [\![2,n]\!]} \{OPT[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1)\}.$$

Algorithme pour le voyageur de commerce

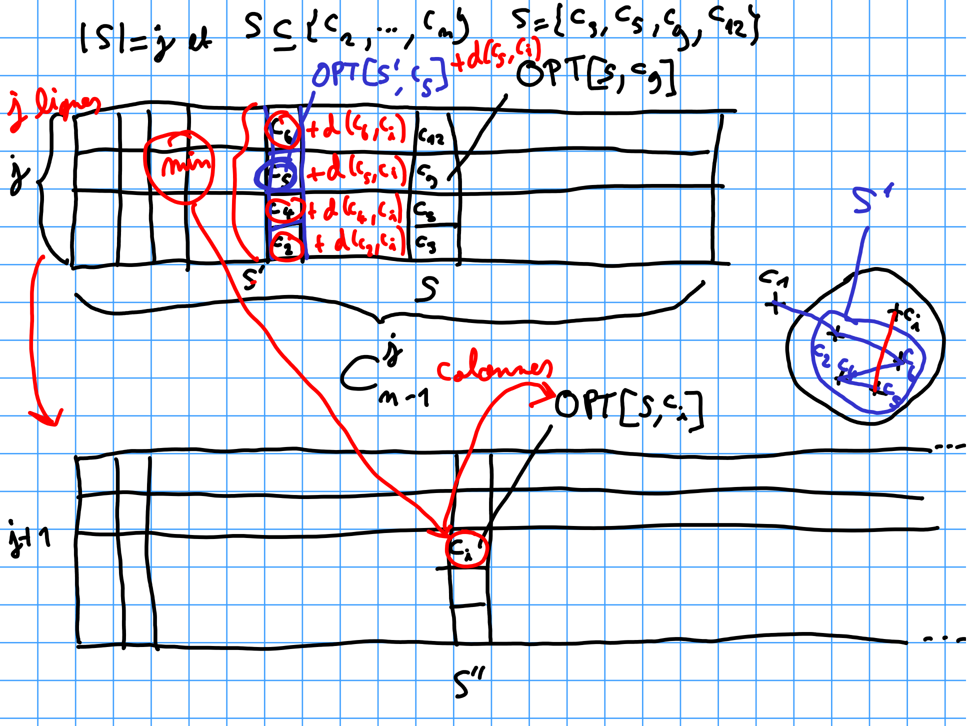
Algorithme 1 : Algorithme pour TSP

1 **pour** i *de* 2 *a* n **faire**
2 | $OPT[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$;
3 **fin**
4 **pour** j *de* 2 *a* $n-1$ **faire**
5 | **pour** *tous les* $S \subseteq \{c_2, \ldots, c_n\}$ *avec* $|S| = j$ **faire**
6 | | **pour** *tous les* $c_i \in S$ **faire**
7 | | | $OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$
8 | | **fin**
9 | **fin**
10 **fin**
11 **retourner** $\min_{i \in [\![2,n]\!]} \{OPT[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1)\}$;

Algorithme pour le voyageur de commerce

Algorithme 1 : Algorithme pour TSP

1 **pour** i *de* 2 *a* n **faire**
2 $\quad OPT[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$;
3 **fin**
4 **pour** j *de* 2 *a* $n-1$ **faire**
5 $\quad$ **pour** *tous les* $S \subseteq \{c_2, \ldots, c_n\}$ *avec* $|S| = j$ **faire**
6 $\quad\quad$ **pour** *tous les* $c_i \in S$ **faire**
7 $\quad\quad\quad OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$
8 $\quad\quad$ **fin**
9 $\quad$ **fin**
10 **fin**
11 **retourner** $\min_{i \in [\![2,n]\!]} \{OPT[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1)\}$;


|S|=j et S ⊆ {c2, ..., cn}
S={c3, c5, c9, c12}
OPT[S', c5]
+d(c5,ci)
OPT[S, c9]
j lignes
j
min
c6 +d(c6,ci)
c5 +d(c5,ci)
c4 +d(c4,ci)
c2 +d(c2,ci)
c12
c9
c5
c3
S'
S
C^j_{n-1} colonnes
OPT[S, ci]
S'
c1
ci
c2
c4
c6
c5
j+1
ci
S''

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ?

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n -$ [illegible]

$2^{n-1} - n$

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n-1)$
- **Total :** $O(n^2 \cdot 2^n) = O^*(2^n)$ ... beaucoup mieux que $O(n!)$ $\sim n^n$

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n-1)$
- **Total :** $O(n^2 \cdot 2^n) = O^*(2^n)$ ... beaucoup mieux que $O(n!)$

Complexite spatiale : $O(n2^n)$ $2^{n-1} \times n$

- le tableau $OPT[S, c_i]$ a une case pour chaque couple (S, c_i)

Analyse de la complexite

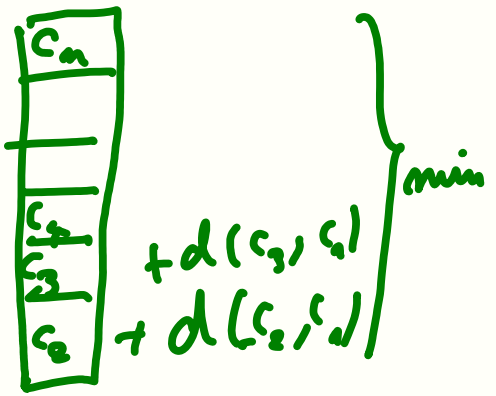
Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n-1)$
- **Total :** $O(n^2 \cdot 2^n) = O^*(2^n)$ ... beaucoup mieux que $O(n!)$

Complexite spatiale : $O(n2^n)$???

- le tableau $OPT[S, c_i]$ a une case pour chaque couple (S, c_i)
- mais... quand on en est a j, on peut ne conserver que les S de taille j et $j-1$

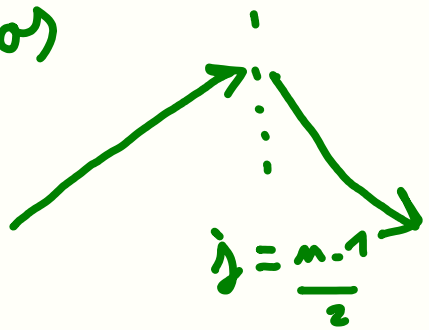
$S = \{c_2, \dots, c_{n-1}\}$



$S = \{c_2, \dots, c_n\}$

C_{n-1}^{j} le plus gros

$C_n^{n/2} \sim \Theta(2^n)$



Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n-1)$
- **Total :** $O(n^2 \cdot 2^n) = O^*(2^n)$ ... beaucoup mieux que $O(n!)$

Complexite spatiale :

- le tableau $OPT[S, c_i]$ a une case pour chaque couple (S, c_i)
- mais... quand on en est a j, on peut ne conserver que les S de taille j et $j-1$
- re-mais... ca fait quand meme $\Omega(2^n)$ pour $j = n/2$.

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n-1)$
- **Total :** $O(n^2 \cdot 2^n) = O^*(2^n)$ ... beaucoup mieux que $O(n!)$

Complexite spatiale :

- le tableau $OPT[S, c_i]$ a une case pour chaque couple (S, c_i)
- mais... quand on en est a j, on peut ne conserver que les S de taille j et $j-1$
- re-mais... ca fait quand meme $\Omega(2^n)$ pour $j = n/2$.
- **au pire de l'algo :** espace $\Omega(n \cdot 2^n)$... c'est la ou le bat blaisse

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n-1)$
- **Total :** $O(n^2 \cdot 2^n) = O^*(2^n)$ ... beaucoup mieux que $O(n!)$

Complexite spatiale :

- le tableau $OPT[S, c_i]$ a une case pour chaque couple (S, c_i)
- mais... quand on en est a j, on peut ne conserver que les S de taille j et $j-1$
- re-mais... ca fait quand meme $\Omega(2^n)$ pour $j = n/2$.
- **au pire de l'algo :** espace $\Omega(n \cdot 2^n)$... c'est la ou le bat blaisse

Conclusion : la prog dynamique permet de gagner du temps en consommant de l'espace

Analyse de la complexite

Complexite temporelle :

- pour un S de taille k, les lignes 6 a 8 prennent un temps $O(k^2)$
- combien d'ensemble S a considerer ? $\longrightarrow 2^{n-1} - (n-1)$
- **Total :** $O(n^2 \cdot 2^n) = O^*(2^n)$ ... beaucoup mieux que $O(n!)$

Complexite spatiale :

- le tableau $OPT[S, c_i]$ a une case pour chaque couple (S, c_i)
- mais... quand on en est a j, on peut ne conserver que les S de taille j et $j-1$ Complexité quand tu nous
- re-mais... ca fait quand meme $\Omega(2^n)$ pour $j = n/2$. tiens...
- **au pire de l'algo :** espace $\Omega(n \cdot 2^n)$... c'est la ou le bat blaisse

Conclusion : la prog dynamique permet de gagner du temps en consommant de l'espace $10^9 = 2^{30}$ 2^{34} place mémoire. 2^{40}

Limites : l'espace est aussi une quantite critique dans les ordinateurs (au moins autant que le temps) Instance max en Ram : 34 en disque : 40