

Algorithms

Divide and conquer

On a la relation de récurrence suivante :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) \end{cases}$$

(prix de diviser, prix de combiner)

a est le nombre de sous-problèmes créés à chaque récurrence, et $\frac{n}{b}$ est la taille de chaque sous-problème.

Exemple

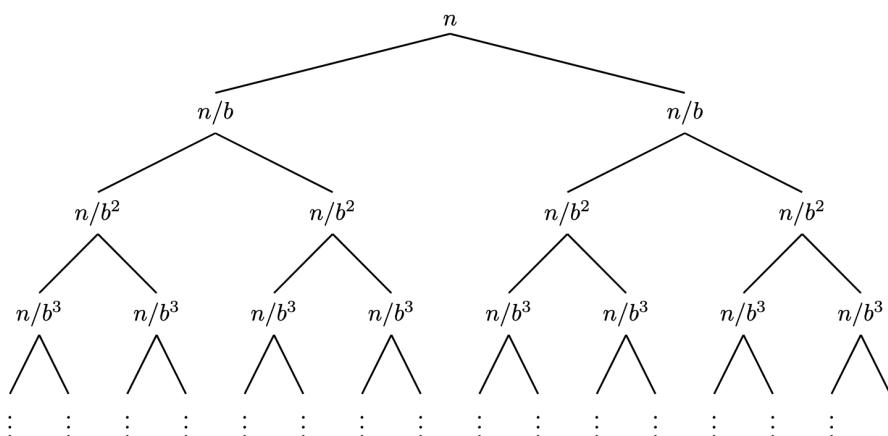
On peut imaginer un algo qui prend une liste de taille n et sépare cette liste en 3 sous-listes de taille $\frac{n}{2}$.

→ $a = 3$ et $b = 2$.

Master theorem

- On est dans le cas **bottom-heavy** quand le coût du divide and combine est plus petit que $n^{\log_b(a-\varepsilon)}$ pour un $\varepsilon > 0$, c-a-d que le nombre total de feuilles. Car (avec de l'analyse), on a $n^{\log_b a} = a^{\log_b n}$, qui est le nombre de feuilles, étant donné qu'on a a enfants par noeud et que la taille de l'arbre est $\log_b(n)$. La majorité du calcul est situé dans le nombre de divisions à faire pour calculer toutes les feuilles (diviser un par deux et combiner deux en un ne coûte pas très cher). Formellement, quand $D(n) + C(n) = O(n^{\log_b(a-\varepsilon)})$. Et donc :

$$T(n) = \Theta(n^{\log_b a})$$



- On est dans le cas **top-heavy** quand le coût du divide et combine est plus grand que $n^{\log_b a + \varepsilon}$. Formellement, quand : $D(n) + C(n) = \Omega(n^{\log_b a})$. C'est-à-dire que diviser et combiner coûte très cher, donc tout le coût du calcul sera fait à la racine (quand on devra séparer la grosse liste de taille n , la racine). Et donc :

$$T(n) = \Theta(f(n))$$

- Quand les deux sont équilibrés, $C(n) + D(n) = \Theta(n^{\log_b a})$, on a :

$$T(n) = \Theta(n^{\log_b a} \cdot \log_b(n))$$

Dans le cas unbalanced ?

Si on a la récurrence suivante : $T(n) = T\left(\frac{n}{5}\right) + 2T\left(\frac{2n}{5}\right) + \Theta(n)$

On sait qu'on aura une branche à gauche très courte en $\log_5(n)$ et une branche à droite très longue en $\log_{\frac{5}{2}}(n)$. On peut donc bound notre taille d'arbre :

$$\log_5(n) \leq \text{hauteur arbre} \leq \log_{\frac{5}{2}}(n)$$

Supposons qu'on est dans le cas d'un arbre balanced avec une hauteur de $\log_5(n)$. On a donc la relation $T_{\text{low}}(n) = 5T\left(\frac{n}{5}\right) + \Theta(n)$. Et on peut construire un arbre balanced avec $T_{\text{high}}(n) = \frac{5}{2}T\left(\frac{2n}{5}\right) + \Theta(n)$. Et d'après le master theorem :

$$\Theta(n \log_5(n)) \leq T(n) \leq \Theta\left(n \log_{\frac{5}{2}}(n)\right) \implies T(n) = \Theta(n \log n)$$

Mais pourquoi peut-on dire que $T_{\text{low}}(n) < T(n)$? Parce que le problème semble très différent, on sépare maintenant en 5 problèmes (donc plus de divisions et de fusions ?), au lieu de 3, et puis on change la taille de ces problèmes, etc. il n'y a pas que la hauteur qui compte pour définir si un problème est plus coûteux qu'un autre, si ?

→ en fait, dans un cas comme dans l'autre, on a **toujours** $\Theta(n)$ travail à chaque étage, et c'est ça qui compte. Certes, on aura plus de divisions et plus de listes à fusionner, mais comme combiner et diviser est fait en $\Theta(n)$, alors diviser deux listes de taille $\frac{2n}{5}$ ou 4 listes de taille $\frac{n}{5}$ est identique (à un facteur constant c devant près). Donc les deux programmes ont le même coût jusqu'à ce qu'un des deux arrive à la fin de sa hauteur (et la hauteur détermine la complexité parce que c'est le nombre d'étages de $\Theta(n)$ qu'on aura, le nombre de cn opérations à faire !).

Résoudre les relations de récurrence

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + c \cdot n \\ &= 2\left(2T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}\right) + c \cdot n = 4T\left(\frac{n}{4}\right) + 2 \cdot cn \\ &\dots = 8T\left(\frac{n}{8}\right) + 3 \cdot cn \end{aligned}$$

On voit un pattern ! $T(n) = 2^k T\left(\frac{n}{2^k}\right) + k \cdot cn$

On en déduit $T(n) = \Theta(n \log n)$

Prouver complexité d'une récurrence par induction

on veut montrer que $\exists a \in \mathbb{R}$ s.t. $a > 0$ and $T(n) \leq a \cdot n \log n$.

- cas de base : on trouve la valeur de $T(2), T(3), T(4)$ et on ajuste a tel que le cas de base soit satisfait.
- récurrence : $2T\left(\frac{n}{2}\right) + cn \leq 2 \cdot \frac{an}{2} \log\left(\frac{n}{2}\right) + c \cdot n = a \cdot n \log\left(\frac{n}{2}\right) + cn = a \cdot n \log(n) - an + cn$ (on utilise l'hypothèse d'induction) $\leq a \cdot n \log(n)$ (on prend $a \geq c$ comme ça $-an + cn \leq 0$).
- donc on peut trouver un a tel quel le cas de base et la récurrence soient satisfait donc $T(n) = O(n \log n)$.

On doit prouver la même chose pour la lower bound ($\exists b > 0$ s.t. $T(n) \geq b \cdot n \log n \forall n \geq 0$).

Matrix multiplication

On a comme entrées deux matrices carrées, $n \times n$:

- $A = (a_{ij})$
- $B = (b_{ij})$

On sort une matrice carrée $n \times n$: $C = (c_{ij})$ où $A \cdot B = C$.

Example (n = 2)

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & b_{11} \\ a_{21} & b_{21} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + \dots + a_{1n}b_{n1} = \sum_{k=1}^n a_{1k}b_{k1}$$

Plus généralement, $c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$

On peut écrire un algo simple qui en temps $\Theta(n^3)$ qui calcule c_{ij} (trois boucles for jusqu'à n qui pour chaque entrée i, j somme tous les produits).

→ le temps utilisé par cet algo est $\Theta(n^2)$, parce qu'on ne crée par une variable à chaque boucle (pour la dernière boucle qui somme les produits on les ajoute à une variable existante c_{ij}).

Avec divide and conquer

Block multiplication : si on sépare notre matrice carré de taille $n \times n$ en 4 matrices de taille $\frac{n}{2} \times \frac{n}{2}$, on peut multiplier ces matrices entre elles pour obtenir la matrice finale. Par exemple :

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

...

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

Comme on le voit, recombiner les problèmes est plus facile (c'est une addition sur tous les i, j donc en n^2). On a 8 sous-produits à calculer.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

avec le Master theorem, on réalise qu'on résout la récurrence : $T(n) = \Theta(n^3)$. pas de progrès !

problème : on a trop de sous-problèmes (8 !)

⌚ Sous-problème plus simple, multiplier des nombres complexes

$$(a + ib) \cdot (c + id) = (a \cdot c - b \cdot d) + i(a \cdot d + b \cdot c) = r$$

Pour multiplier 2 complexes, on a dû faire 4 produits de nombres réels.

Calculons :

$$s_1 = (a + b) \cdot (c + d) = ac + ad + bc + bd$$

$$s_2 = a \cdot c$$

$$s_3 = b \cdot d$$

$$r = (s_2 - s_3) + i(s_1 - s_2 - s_3)$$

Maintenant on a un produit de moins nécessaire pour trouver r ! On a plus d'additions et de soustractions. On peut utiliser le même principe pour les matrices.

En appliquant cette méthode aux matrices, on arrive à 7 produits !

Multiplier deux nombres entiers

On a deux entiers de n chiffres x et y en base b . On doit trouver $x \cdot y$.

Ce qu'on a vu au lycée c'est un algo en $\Theta(n^2)$.

Note : multiplier par b^k c'est considéré en $O(1)$, c'est juste un shift.

On peut séparer nos deux nombres en deux parties :

$$X = X_H \cdot 10^{\frac{n}{2}} + X_L \quad \text{et} \quad Y = Y_H \cdot 10^{\frac{n}{2}} + Y_L$$

Par exemple si $X = 1234$, on a $\frac{n}{2} = 2$ donc $X_H = 12, X_L = 34$.

On veut calculer :

$$\begin{aligned} X \cdot Y &= (X_H \cdot 10^{\frac{n}{2}} + X_L) \cdot (Y_H \cdot 10^{\frac{n}{2}} + Y_L) \\ &= X_H \cdot Y_H \cdot 10^n + (X_H Y_L + X_L Y_H) \cdot 10^{\frac{n}{2}} + X_L Y_L \end{aligned}$$

On a donc 4 multiplications de taille $\frac{n}{2}$! mais avec l'algo de **Karatsuba**, on peut réutiliser l'astuce des nombres complexes et calculer :

$$M = (X_H + X_L) \cdot (Y_H + Y_L)$$

$$X \cdot Y = X_H Y_H \cdot 10^n + (M - X_H Y_H - X_L Y_L) \cdot 10^{\frac{n}{2}} + X_L Y_L$$

En fait, on a donc en fait que 3 multiplications de taille $\frac{n}{2}$ à calculer ! $M, X_L Y_L, X_H Y_H$

On arrive à :

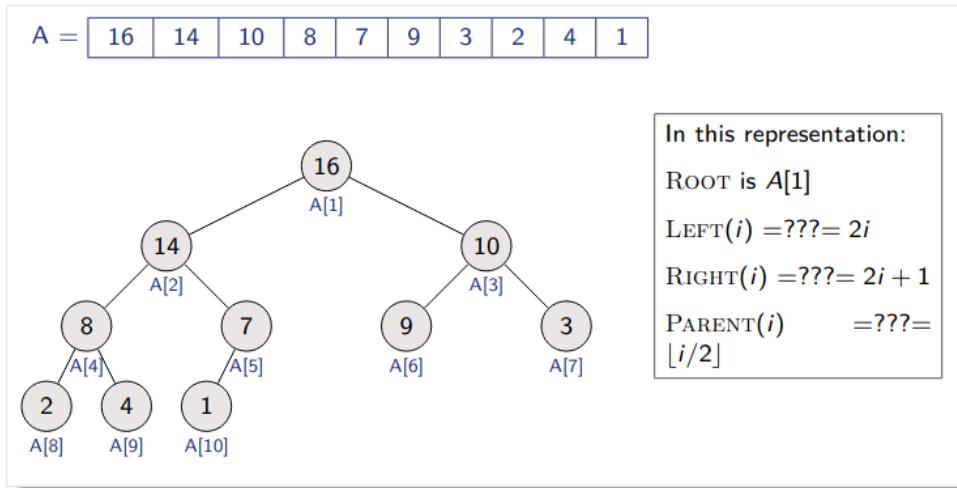
$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) = \Theta\left(n^{\log_2(3)}\right)$$

Heaps

(min/max)-heap property : on veut que chaque noeud ait une valeur plus grande (ou plus petite pour le min-heap) que chacun de ses enfants directs.

- on sait que la maximum est toujours en haut de l'arbre
- la hauteur d'un noeud est le plus long chemin simple (le nombre de segments) entre le noeud et une feuille

On utilise un tableau pour stocker les heaps :



Heapify

Comment maintenir la **(min/max)-heap property** ? Il existe un algorithme qui nous permet de retrouver la heap-property si elle n'est pas respectée qu'à la racine en **O(log h)**.

- comparer $A[i], A[L(i)], a[R(i)]$
- si nécessaire, échanger $A[i]$ avec le plus grand des enfants
- continuer jusqu'à ce que la règle soit correcte

Build Max/Min Heap

Cet algorithme est en $\Theta(n)$ et permet de construire un heap à partir d'un tableau désordonné. Il appelle **heapify** sur chaque élément du tableau (sauf sur les tous petits noeuds du bas mais ça importe peu). $n \cdot \text{heapify} = n \cdot \log h \dots ?$

Sauf qu'en fait, plus on remonte dans l'arbre pour faire n'est pas $\log n$!

$$\sum_{h=0}^{\log n} (\# \text{ noeuds}) \cdot O(h) = O\left(\sum_{h=0}^{\log n} \frac{n}{2^h} \cdot h\right)$$

(le nombre de noeuds à chaque hauteur h est inférieur à $\frac{n}{2^h}$)

On peut montrer avec de l'analyse que

$$\sum_{h=0}^{\log n} \frac{h}{2^h} < 2 \text{ car } \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = 2$$

```
build_max_heap(A, n):
    for i = floor(n/2) downto 1
        max_heapify(A, i, n)
```

⌚ Comment montrer que build_max_heap' est en $n \log n$

BUILD-MAX-HEAP'(A)

1. $A.\text{heap-size} = 1$
2. **for** $i = 2$ **to** $A.\text{length}$
3. MAX-HEAP-INSERT($A, A[i]$)

On sait que pour chaque noeud i inséré on a un travail de $\log(i)$ pour le faire remonter dans le worst case :

$$\sum_{i=0}^h \text{floor}(\log(i)) \leq \sum_{i=\frac{n}{2}}^n \text{floor}(\log(i)) \leq \frac{n}{2} \log\left(\frac{n}{2}\right) \text{ en } O(n \log n)$$

Heap sort

Heapsort a la même complexité que le **merge sort**, mais est **in-place** (comme insertion sort). Le meilleur des deux mondes ?

- on crée un max-heap à partir du tableau
- on fait un **build-max-heap** ($n \log n$)

- on prend le premier élément max et on le met au début du tableau (on le discard)
- on met le dernier élément du tableau à sa place
- on appelle **heapify**
- etc. en boucle jusqu'à ce que le tableau soit trié (donc n heapify, en $n \log n$)

```
heap_sort(A, n):
    build_max_heap(A, n)
    for i = n downto 2
        exchange A[1] with A[i]
        max_heapify(A, 1, i - 1)
```

Priority Queues

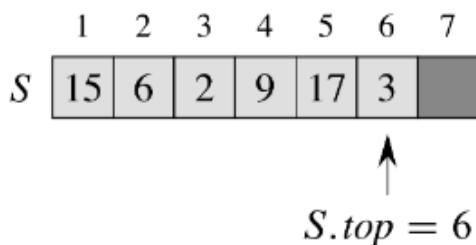
- we want to maintain a **dynamic set S of elements** using heaps
- each set element has a **key** and a **value**

Supported operations

- `maximum_element(A, n)` : this is $\Theta(1)$, we just have to return the first one
- `extract_maximum_element(A, n)` : $\Theta(\log n)$, we remove the maximum element, and replace it with the last one and run `heapify`
- `increase_key_value(A, key, n)` : $\Theta(\log n)$, on met à jour l'élément puis on regarde tous ses parents. S'il est plus grand, on inverse les deux.
- `insert_into_heap(A, key, n)` : $\Theta(\log n)$, on insert l'élément tout au bout avec une valeur de $-\infty$ puis on appelle `increase_key_value` dessus

Stack implementation (last-in, first-out)

Utiles pour les allocations mémoires.



`Q.top` pointe à la position du dernier élément (celui qui vient d'arriver).

Opérations

- `empty` : on check si le top est 0
- `push` : $S.top \leftarrow S.top + 1$ puis $S[S.top] \leftarrow x$

- `pop : S.top <- S.top - 1 et return S[S.top + 1]`

elles sont en $\Theta(1)$.

Queue implementation (first-in, first-out)

- `Q.head` pointe à la position du premier élément
- `Q.tail` pointe à la position de là où le prochain élément arriverait

Opérations

- `enqueue(Q, x) :`
`Q[Q.tail] = x et if (Q.tail == Q.length) Q.tail = 1 else Q.tail += 1`

parce que si on enqueue deux fois, on va faire pointer la tail vers 1 ! (parce que 13 n'existe pas)

et comme la head est 7 (soit après), on sait qu'on aura 8, 9, etc. définis

Stacks aned et sont bien et très performantes mais... ont un support limité : pas de recherche, par exemple.

Linked list

- `L.head` pointe vers la tête de la liste
- chaque noeud `N` stocke `N.prev`, `N.key`, `N.next`

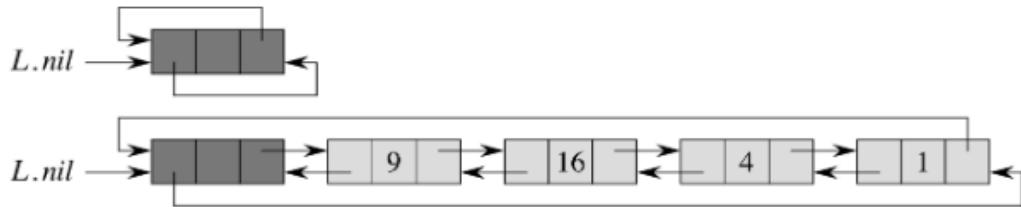
Opérations

- `search : $\Theta(n)$`
`x <- L.head and while(x != nil and x.key != k) x = x.next then return x`
- `insert(x) : $\Theta(1)$`
`new_el = (/, x, L.head) and L.head.next = L.head and L.head <- new_el`
- `delete(x) : $\Theta(1)$`
`x.prev.next = x.next (on lit celui avant x à celui après x)`
 (attention à bien gérer tous les cas)

💡 Pour simplifier les opérations (gérer les cas de nuls, etc), on peut ajouter des sentinels!

C'est-à-dire ajouter des éléments fake pour que les éléments réels ne soient jamais les premiers ou derniers.

Sentinels



LIST-DELETE(*L,x*)

1. **if** *x.prev* ≠ *nil*
2. *x.prev.next* ← *x.next*
3. **else** *L.head* ← *x.next*
4. **if** *x.next* ≠ *nil*
5. *x.next.prev* ← *x.prev*

simplified

LIST-DELETE'(*L,x*)

1. *x.prev.next* ← *x.next*
2. *x.next.prev* ← *x.prev*

Binary search trees

à gauche, on met tous les nombres plus petits que la racine, et à droite tous les nombres plus grands (et on préfère avoir un arbre équilibré). créer un binary search trees c'est comme encoder dans un arbre la stratégie pour trouver le nombre avec un jeu de "tu dis plus grand ou plus petit"

Comment stocker un binary search tree ?

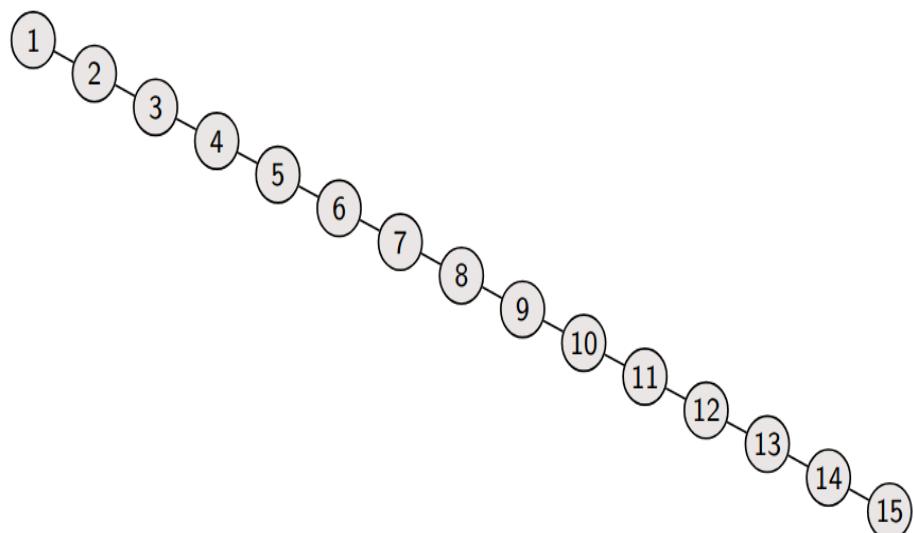
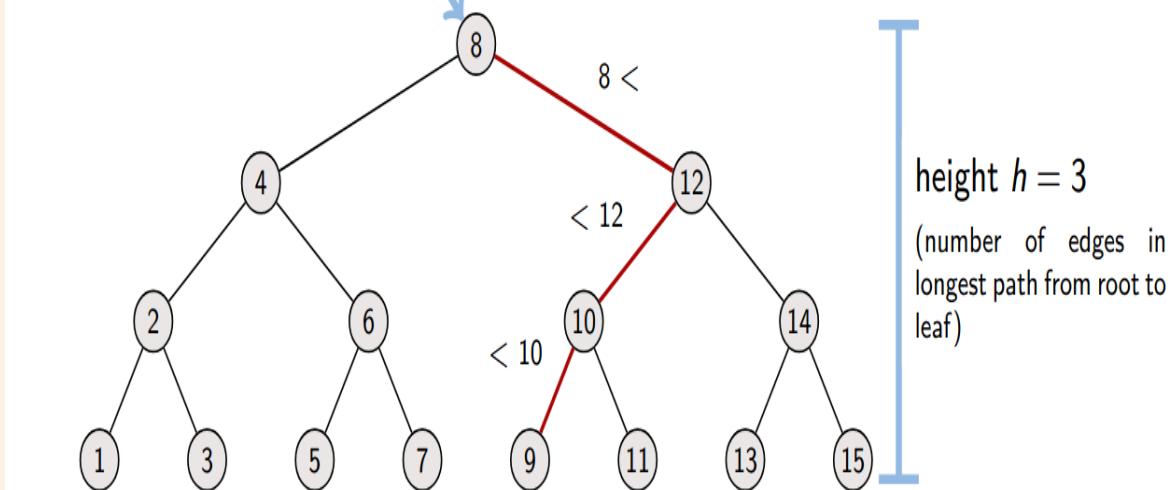
On peut faire quelque chose comme les linked list. Chaque élément stocke :

- la référence du parent *x.p*
- la référence de l'enfant gauche *x.left*
- la référence de l'enfant droit *x.right*
- la valeur de l'élément *x.key*

② Pourquoi on n'utilise pas le même tableau que pour les heaps ?

Pour les heaps, utilise le fait qu'on a à chaque fois un arbre "nearly-completed" (équilibré). Ici, on ne force pas forcément ça, on veut juste que l'enfant droit soit plus grand que le parent, et que l'enfant gauche soit plus petit. Ces deux exemples sont des binary search trees valides :

Tree T has a root: $T.\text{root}$



Opérations de recherche

La propriété la plus importante est la taille de l'arbre h .

- `tree_search(ptr_to_root, key)` en $\Theta(h)$

```

if (x == nil or ptr_to_root.key == key) return x
else if (k < ptr_to_root.key) return tree_search(ptr_to_root.left,
key)
else return tree_search(ptr_to_root.right, key)
  
```

- `tree_minimum(ptr_to_root)` en $\Theta(h)$

```
c while (ptr_to_root.left != nil) ptr_to_root = ptr_to_root.left return
```

- `ptr_to_root`
- same for `tree_maximum`
- `successor(ptr_to_root)` (the next bigger node)

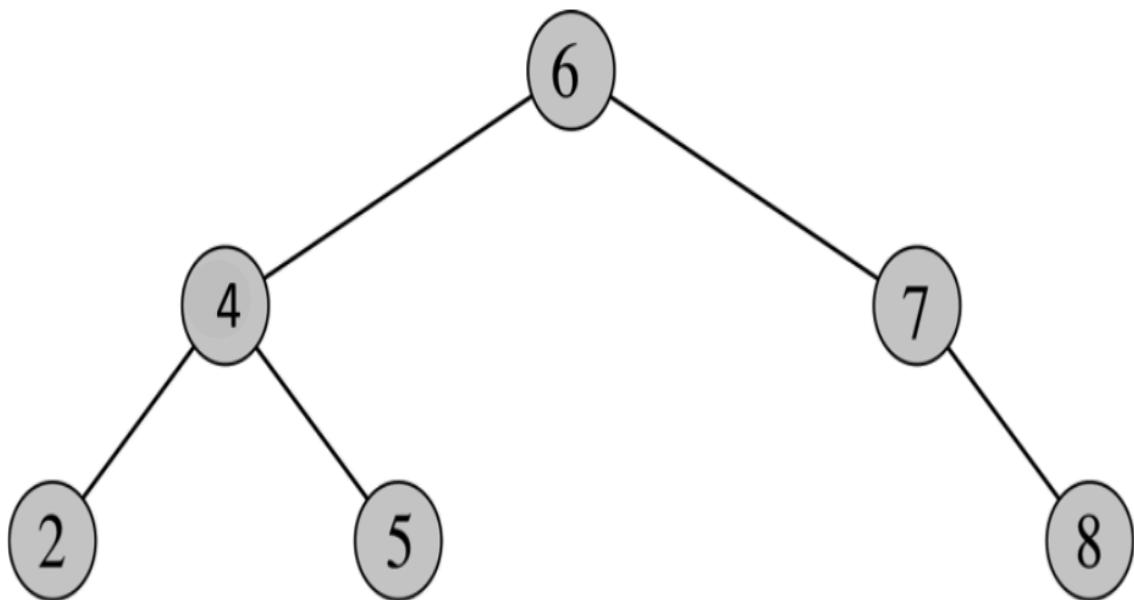
```

if (ptr_to_root.right != null) return tree_minimum(ptr_to_root.right)
else
    y = ptr_to_root.p
    while (y != null && x == y.right)
        x = y
        y = y.p
    return y

```

② successor ?

C'est le noeud juste plus grand que le noeud actuel. Ici, le successeur de 5 est 6 :



printing orders :

- in order : afficher à gauche, puis le root, puis à droite
- pre order : afficher le root, puis à gauche, puis à droite
- post order : afficher à gauche, puis à droite, puis le root

comment insérer dans un binary search?

- on cherche pour la clef
- quand on trouve nil, on insert là

comment supprimer z ?

- si c'est une feuille, on supprime
- si c'est un noeud avec un enfant, on fait comme pour une linked list (on change juste les références)
- sinon, on trouve le successeur y et on remplace z par y

Dynamic programming

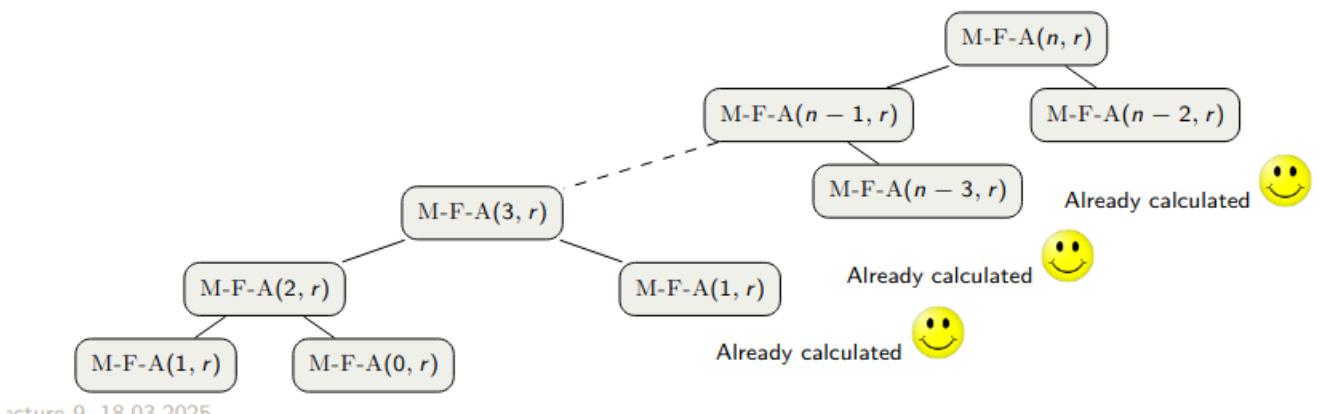
Se souvenir de se qu'on a fait pour éviter de le refaire.

Bottom-up : on part de $f(0)$ et on remonte en cachant les résultats (fibo). ça ressemble un peu à remplir une table au fur et à mesure.

Top-down fibo : on part toujours de $f(n)$ puis on appelle $f(n - 1)$, $f(n - 2)$, avec un dictionnaire mémo qu'ils mettent à jour. on part de l'hypothèse qu'on connaît les réponses des problèmes précédents.

```
def fibonacci(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
    return memo[n]

# Example usage:
print(fibonacci(10)) # Output: 55
```



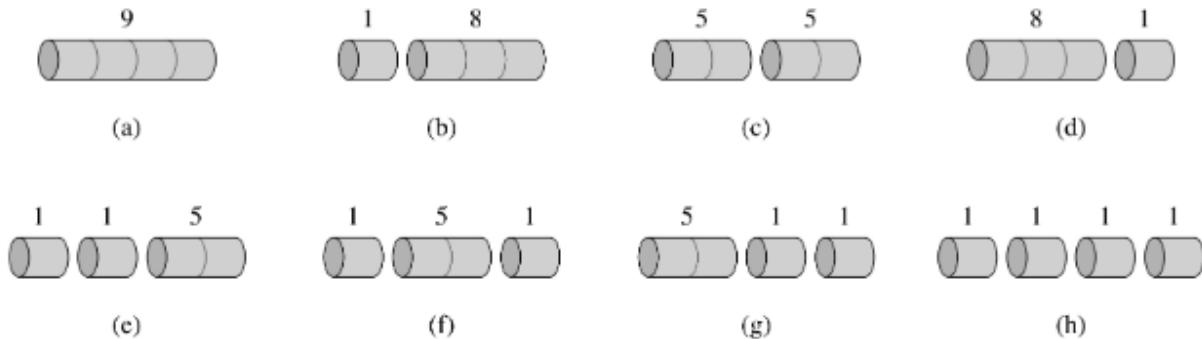
Lecture 9, 18.03.2025

Rod cutting

Entrées : une metal rod de taille n , une table des prix p_i pour des rod de tailles i

Sortie : décider comment couper la rod en pièces et maximiser le prix

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Pour $n = 1 : r(X) = 1$

Pour $n = 2 : r(X X) = 2, r(XX) = 5$

Pour $n = 3 : r(XX X) = 6, r(X X X) = 3, r(XXX) = 8$

On sait que si on décide de couper la tige de longueur n à l'indice i , on a :

$$\text{Optimal}(n) = p(i) + \text{Optimal}(n - i)$$

Mais comment savoir si le i est optimal ? On teste tout !

$$r(n) = \max_{i=1, \dots, n-1} \{p(i) + r(n - i); p(n)\}$$

- $r(n)$, représente le revenu maximal que l'on peut obtenir avec une tige de longueur n .
- $p(i)$ est le prix d'un morceau de longueur i selon la table des prix donnée.
- $r(n - i)$ correspond au revenu maximal obtenu avec le reste de la tige (de longueur $n - i$)

```
def rod_cut(n, prices):
    if n == 0:
        return 0
    max_value = -inf
    for i in range(1, n + 1):
        max_value = max(max_value, prices[i] + rod_cut(n - i, prices))
    return max_value

prices = [0, 1, 5, 8, 9] # Indexation des prix (0 pour aligner)
print(rod_cut(4, prices)) # Devrait afficher 10
```

bad, it's slow

top-down memoized

```

def rc_memo(n, prices, memo=None):
    if memo is None:
        memo = [-inf] * (n + 1)

    if n == 0:
        return 0
    if memo[n] >= 0:
        return memo[n]

    q = -inf
    for i in range(1, n + 1):
        q = max(q, prices[i] + rc_memo(n - i, prices, memo))

    memo[n] = q
    return q

```

bottom-up memoized

```

def rc_bottom_up(n, prices):
    dp = [0] * (n + 1)
    for j in range(1, n + 1):
        q = -inf
        for i in range(1, j + 1):
            q = max(q, prices[i] + dp[j - i])
        dp[j] = q
    return dp[n]

```

Équivalence de deux algos

$$r_2(n) = \max_{i=1,\dots,n-1} \{r(n-i) + p(i); p(n)\}$$

est équivalent à

$$r_1(n) = \max_{i=1,\dots,n-1} \{r(n-i) + r(i); p(n)\}$$

on voit assez vite que $r_1(n) \geq r_2(n)$ (au lieu de vendre un chunk de taille i on prend la meilleure façon de le vendre)

$$i = \arg \max_i \{r(n-i), r(i)\}$$

Change coin making

ça ressemble au problème du rod-cutting

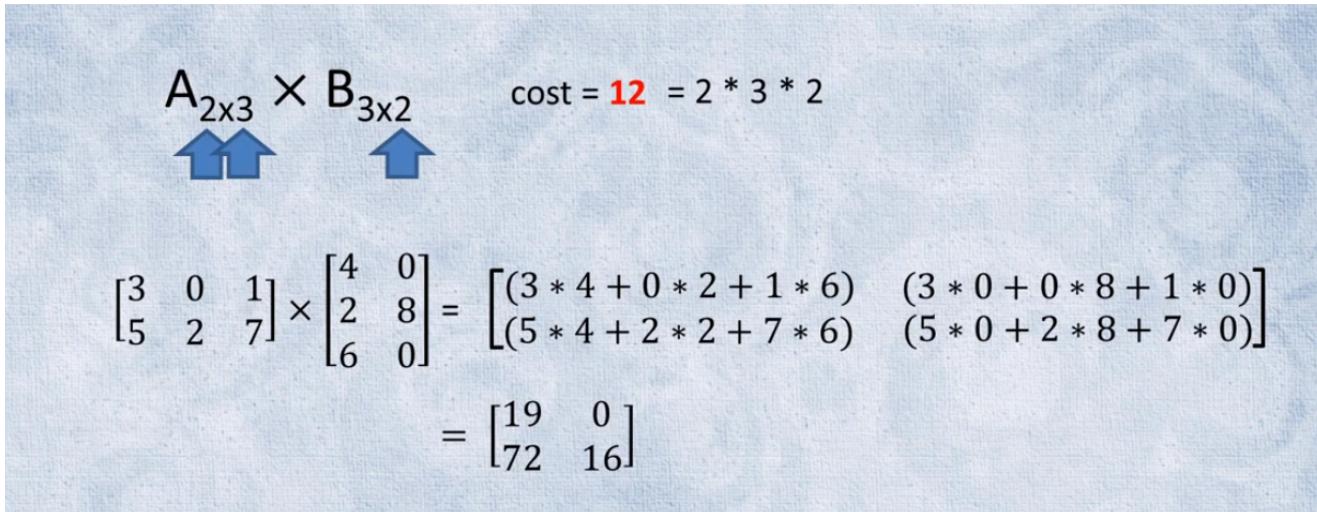
Matrix-chain multiplication

Comment les faire plus rapidement quand ils sont chaînés ?

On veut multiplier $A_1 (50 \times 5)$, $A_2 (5 \times 100)$, $A_3 (100 \times 10)$ (ce ne sont pas des matrices carré).

Nombre d'opérations pour $A_1 A_2 : 50 \times 5 \times 100$
 pour $B A_3 : 50 \times 100 \times 10$
 somme : 75 000

Sinon on peut faire $A_1 (A_2 A_3) : 5 \cdot 100 \cdot 10 + 50 \cdot 5 \cdot 10 = 7500$ multiplications



$$\begin{matrix}
 A_{2 \times 3} & \times & B_{3 \times 2} \\
 \uparrow & & \uparrow \\
 \begin{bmatrix} 3 & 0 & 1 \\ 5 & 2 & 7 \end{bmatrix} & \times & \begin{bmatrix} 4 & 0 \\ 2 & 8 \\ 6 & 0 \end{bmatrix} = \begin{bmatrix} (3 * 4 + 0 * 2 + 1 * 6) & (3 * 0 + 0 * 8 + 1 * 0) \\ (5 * 4 + 2 * 2 + 7 * 6) & (5 * 0 + 2 * 8 + 7 * 0) \end{bmatrix} \\
 & & = \begin{bmatrix} 19 & 0 \\ 72 & 16 \end{bmatrix}
 \end{matrix}$$

Algorithme

Entrée : une suite de n matrices où A_i a comme dimensions $p_{i-1} \cdot p_i$

Sortie : la meilleure façon de mettre les parenthèses pour minimiser les multiplications

On pourrait utiliser un algorithme récursif :

$$\left[A_1 \times A_2 \times A_3 \times A_4 \right] \times \left[A_5 \times \dots A_n \right]$$

i k j

```
MxCost cost(List<MxCost> dims, int i, int j) {
    if (i == j)
        return dims.get(i); // just one matrix, so cost is zero

    MxCost min = null;
    for (int k = i; k < j; k++) {
        MxCost left = cost(dims, i, k);
        MxCost right = cost(dims, k+1, j);
        MxCost result = left.multiply(right);

        if (min == null || min.cost > result.cost)
            min = result; // found a better solution
    }
    return min;
}
```

puis l'améliorer en passant une map solutions :

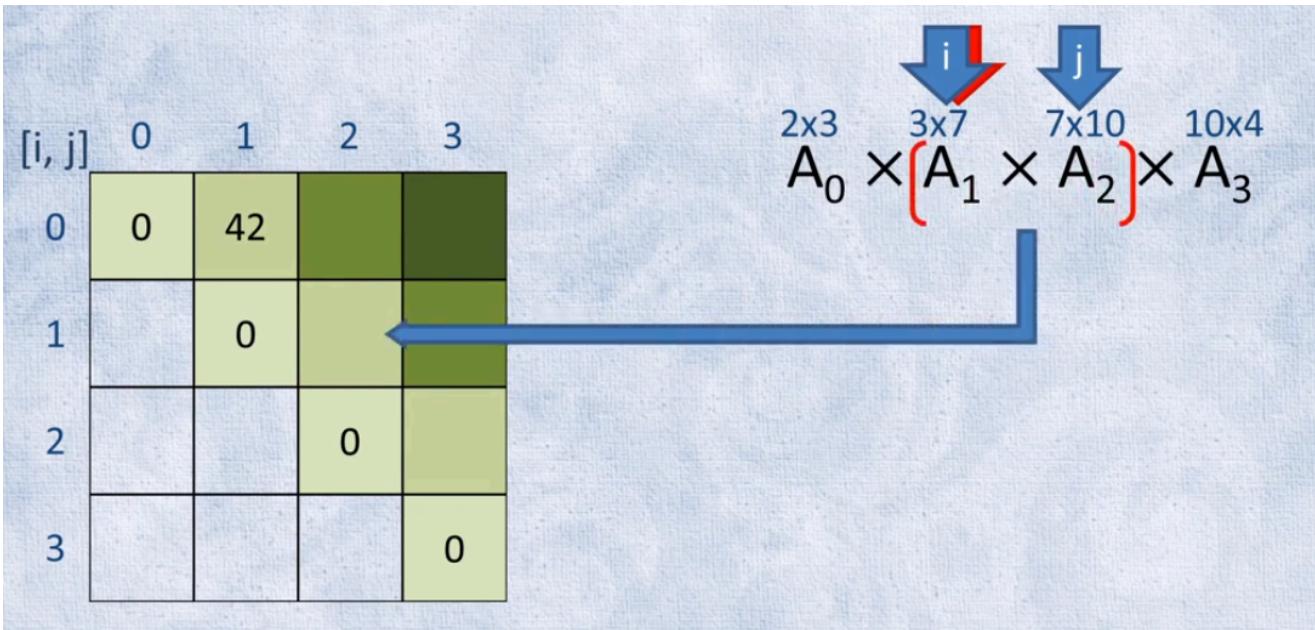
```
MxCost cost(List<MxCost> dims, int i, int j, Map<String, MxCost> solutions) {
    if (i == j)
        return dims.get(i); // just one matrix, so cost is zero

    String key = i + " ." + j;
    MxCost solution = solutions.get(key);
    if (solution != null)
        return solution;
    }

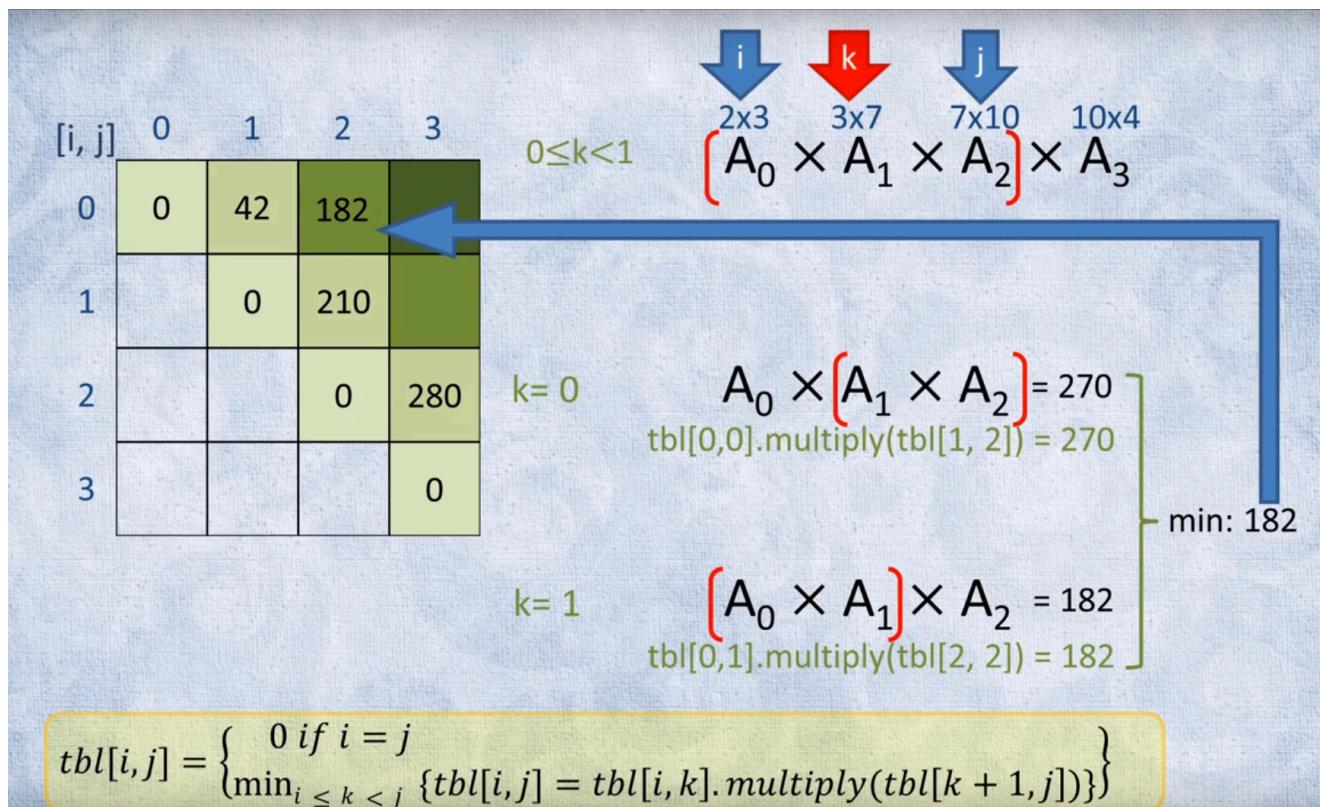
    MxCost min = null;
    for (int k = i; k < j; k++) {
        MxCost leftMx = cost(dims, i, k, solutions);
        MxCost rightMx = cost(dims, k+1, j, solutions);
        MxCost result = leftMx.multiply(rightMx);

        if (min == null || min.cost > result.cost)
            min = result; // found a better solution
    }
    solutions.put(key, min);
    return min;
}
```

Algorithme dynamic programming (bottom up)



on remplit une table comme ça. c'est facile de remplir quand on multiplie deux matrices ensemble mais qu'est-ce qu'il se passe quand on multiplie 3 ensemble ? on doit prendre le minimum.



```

MxCost cost(List<MxCost> dims) {
    int n = dims.size();
    MxCost tbl[][] = new MxCost[n][n];

    for (int i = 0; i < n; i++) {
        tbl[i][i] = dims.get(i); // all have cost of zero along the diagonal
    }

    for (int len = 1; len < n; len++) { // repeat for each sub-sequence length
        for (int i = 0; i < n - len; i++) {
            for (int k = i, j = i + len; k < j; k++) { // check for minimum cost

                MxCost temp = tbl[i][k].multiply(tbl[k+1][j]);
                if (tbl[i][j] == null || temp.cost < tbl[i][j].cost) {
                    tbl[i][j] = temp;
                }
            }
        }
    }

    return tbl[0][n-1];
}

```

Longest common sub-sequence

Input : 2 séquences $X = (x_1, \dots, x_m)$ et $Y = (y_1, \dots, y_n)$.

Output : une sous-séquence commune aux deux dont la longueur est maximale.

La sous-séquence n'a pas besoin d'être consécutive, mais elle doit être dans l'ordre.

Approche naïve : tester toutes les sous-séquences possibles dans X et vérifier si elle existe dans Y . On en aurait pour 2^n (nb de possibilités) $\cdot n$ (tester si elle existe dans Y).

On prend le problème entre deux variables i, j

$LCS(X_i, Y_j)$?

avec $X_i = < x_1, \dots, x_i >$ et $Y_j = < y_1, \dots, y_j >$

On compare la dernière lettre :

- case $x_i = y_j$, on prend le maximum entre :
 - $LCS(x_{i-1}, y_j)$
 - $LCS(x_i, y_{j-1})$
 - $1 + LCS(x_{i-1}, y_{j-1})$
- case $x_i \neq y_j$, on prend le maximum entre :
 - $LCS(x_{i-1}, y_j)$
 - $LCS(x_i, y_{j-1})$
- Si $Z = < z_1, \dots, z_k >$ est la LCS des deux, alors si $x_i = y_j$ alors $z_k = x_i = y_j$ (sinon on pourrait améliorer la chaîne en rajoutant x_i à la fin et ce serait tjs une subsequence).
- Si $x_i \neq y_j$ alors $z_k \neq x_i$ et Z est un LCS de X_{i-1} et Y_j .
- Si $x_i \neq y_j$ alors $z_k \neq y_j$ et Z est un LCS de X_i et Y_{j-1} .

Analyse de la complexité

- compter le nombre de cellules à remplir $m \cdot n$
- compter le temps de remplir une cellule $\Theta(1)$.

$\rightarrow \Theta(m \cdot n)$

LCS(X, Y, m, n):

```

let b[0, ..., m][0, ..., n] and c [0, ..., m][0, ..., n] be new tables

for i= 1...m:
    c [i][0] = 0

for j=1...n:
    c [0][j] = 0

for i =1..m:
    for j=1..n:
        if X[i] == Y[j]: # on comp les dernières lettres
            c[i][j] = c[i - 1][j - 1] + 1
            b[i][j] = "arrow_diago"
        else
            if c[i - 1][j] >= c[i, j-1]
                c[i][j] = c[i - 1][j]
                b[i][j] = "arrow_up"
            else
                c[i][j] = c[i][j-1]
                b[i][j] = "arrow_left"
return (c, b)

```

Optimal binary search trees

Entrée : un ensemble de clefs triées, et une probabilité p_i que la clef k_i soit cherchée.

Sortie : un BST qui minimise le coût de recherche

Graphs

$G = (V, E)$ avec un ensemble de vertices (points) V et un ensemble de segments (edges) entre les deux E .

Undirected graph

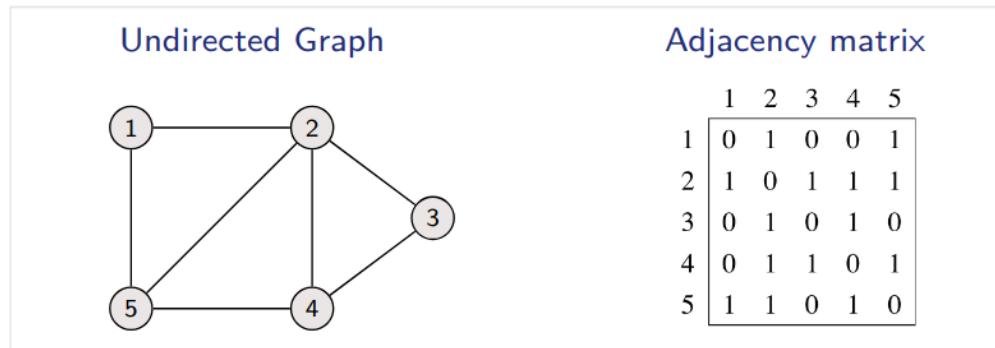
Comment stocker un graph ? On peut les stocker dans une adjancy matrice $V \times V$ où chaque élément est un 0 ou un 1 en fonction de si le lien entre les deux noeuds est fait.

Espace : $\Theta(V^2)$

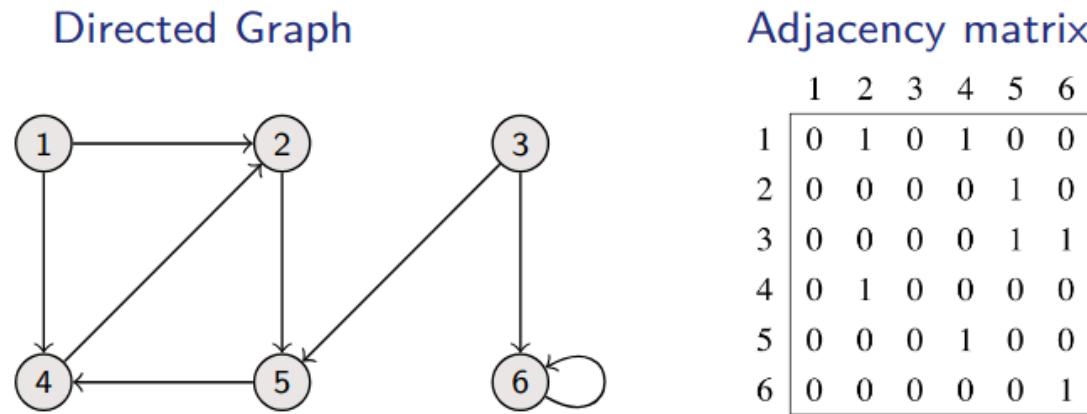
Temps pour lister tous les noeuds adjacents à u : $\Theta(V)$

Temps pour déterminer si $(u, v) \in E$: $\Theta(1)$

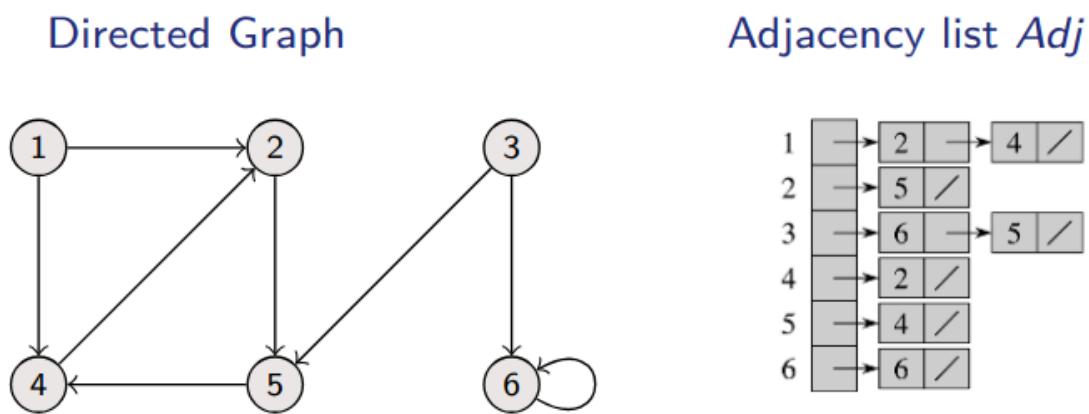
→ pas efficace en termes d'espace



Directed graph



Adjacency list



1 est connecté à 2 et est connecté à 4.

Espace : $\Theta(V + E)$

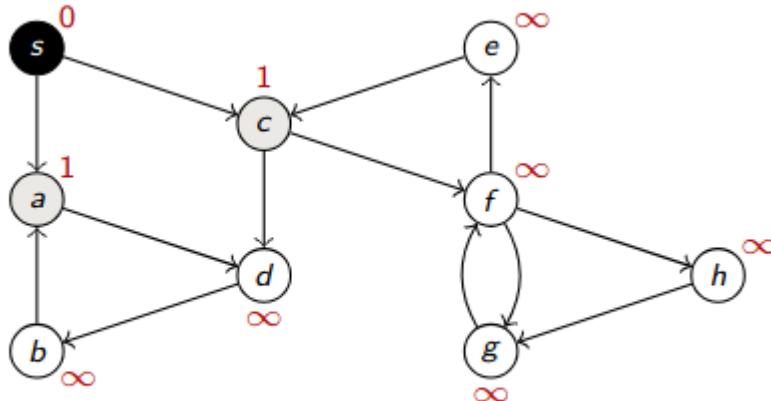
Temps pour lister tous les noeuds adjacents $\Theta(\text{degree}(u))$.

Temps pour déterminer si $(u, v) \in E$: $\Theta(\text{degree}(u))$

Breadth-First search

Entrée : un graphe, un point s et un point v

Sortie : la distance de s à v , pour tous les $v \in V$

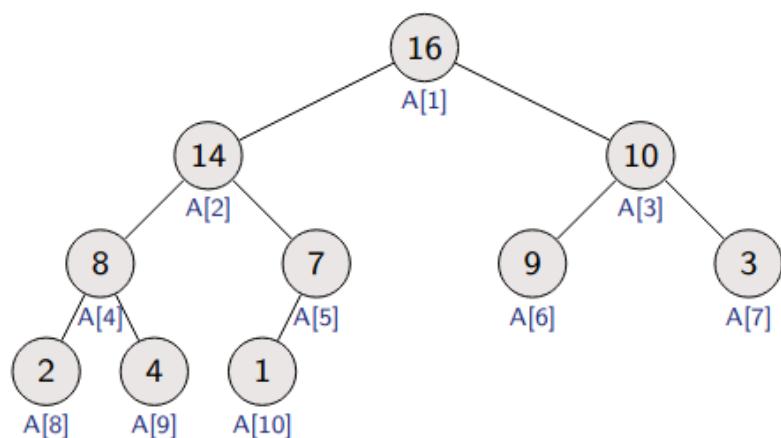


Queue $Q = a, c$

noir : queued + processed

gris : queued

16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---



In this representation:

ROOT is $A[1]$

LEFT(i) = ??? = $2i$

RIGHT(i) = ??? = $2i + 1$

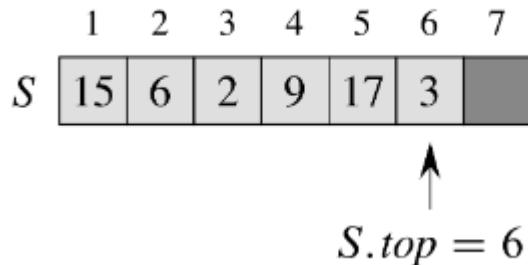
PARENT(i) = ??? = $[i/2]$

Runtime : $O(V + E)$

Lemma : le nombre de personnes qui ont un nb d'amis impairs est pairs (undirected graph)

$$\sum_u \deg(u) = 2 \cdot |E| \Rightarrow \frac{\sum_u \deg(u)}{2} = |E|$$

car chaque arête touche deux sommets



Depth-first search

Idée : on essaye d'aller le plus loin possible (contrairement au BFS où on découvre tous les sommets autour d'abord).

On part de b , on découvre a puis de a on découvre autant que possible, par exemple h (si on garde un ordre alphabétique), puis g , puis on a plus rien à découvrir ! on revient à h et on regarde ce qu'on peut découvrir. On ne peut plus rien découvrir, on revient à a , etc.

Comme pour le BFS, si il nous manque des sommets à explorer quand on a finit, on choisit un autre noeud au hasard et on part de lui.

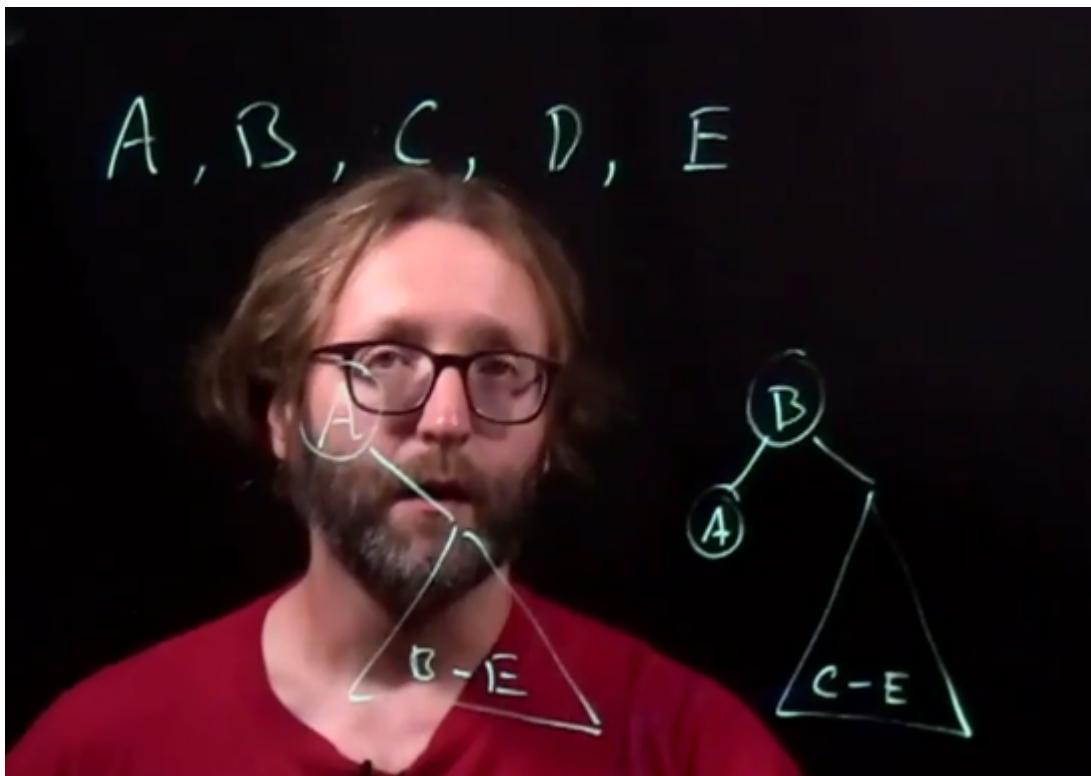
```

DFS( $G$ )
  for each  $u \in G.V$ 
     $u.color = \text{WHITE}$ 
     $time = 0$ 
    for each  $u \in G.V$ 
      if  $u.color == \text{WHITE}$ 
        DFS-VISIT( $G, u$ )
  
```

```

DFS-VISIT( $G, u$ )
   $time = time + 1$ 
   $u.d = time$ 
   $u.color = \text{GRAY}$           // discover  $u$ 
  for each  $v \in G.Adj[u]$     // explore  $(u, v)$ 
    if  $v.color == \text{WHITE}$ 
      DFS-VISIT( $v$ )
   $u.color = \text{BLACK}$ 
   $time = time + 1$ 
   $u.f = time$                 // finish  $u$ 
  
```

Optimal binary search trees



Adjacency list

Space = $\Theta(V + E)$ **Time:** to list all vertices adjacent to u : $\Theta(\text{degree}(u))$ **Time:** to determine whether $(u, v) \in E$: $O(\text{degree}(u))$

Adjacency matrix

Space = $\Theta(V^2)$ **Time:** to list all vertices adjacent to u : $\Theta(V)$ **Time:** to determine whether $(u, v) \in E$: $\Theta(1)$

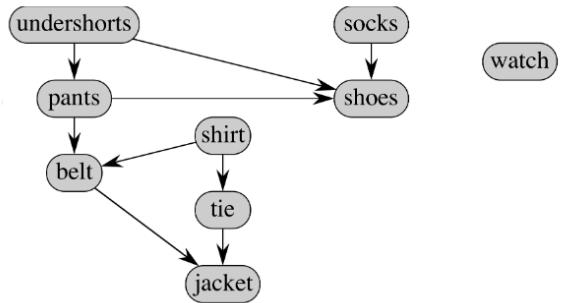
- 1 $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ and neither of u and v are descendant of each other
- 2 $u.d < v.d < v.f < u.f$ and v is a descendant of u
- 3 $v.d < u.d < u.f < v.f$ and u is a descendant of v .

Topological sorting

Entrée : un graphe acyclique (DAG) $G = (V, E)$

Sortie : un ordre linéaire de noeuds t.q $(u, v) \in E$ si u est plus petit que v

Getting dressed in the morning:



in which order?



Graph acylique : pas de back edge après un DFS

TOPOLOGICAL-SORT(G):

1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Do not need to sort by finishing times

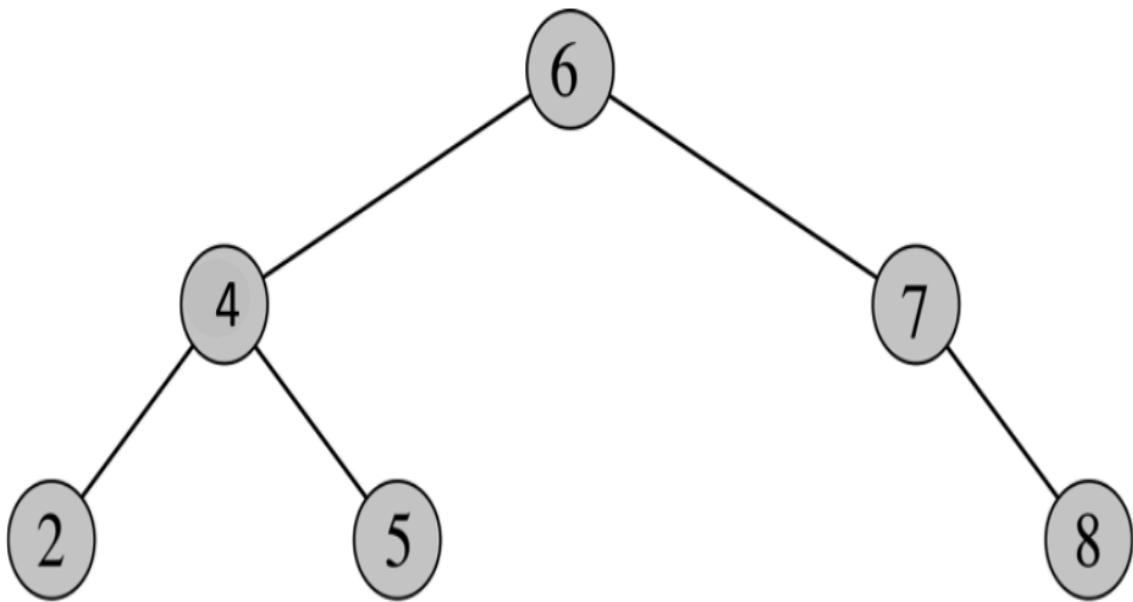
- ▶ Can just output vertices as they are finished and understand that we want the reverse of the list
- ▶ Or put them onto the front of a linked list as they are finished. When done, the list contains vertices in topologically sorted order.

Time: $\Theta(V + E)$ (same as DFS)

Trouver les SCC

- appeler DFS pour calculer les finishing times $u.f$ de tous les noeuds
- calculer G^T (inverser tous les segments)
- appeler G^T mais dans la boucle principale, considérer les noeuds dans un ordre descendant (en fonction des finishing times)
- afficher les noeuds de chaque tree

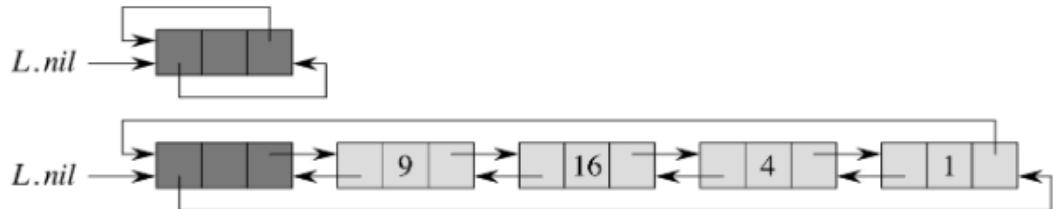
(en fait en inversant les flèches, on évite de partir du strongly connected component + à l'intérieur d'un strongly connected component échanger les flèches ne changent rien à la structure)



Flow networks

On veut un graphe sans "parallel" edges (un qui va de (u, v) , un (v, u)).

Sentinels



`LIST-DELETE(L,x)`

1. **if** $x.prev \neq nil$
2. $x.prev.next \leftarrow x.next$
3. **else** $L.head \leftarrow x.next$
4. **if** $x.next \neq nil$
5. $x.next.prev \leftarrow x.prev$

simplified

`LIST-DELETE'(L,x)`

1. $x.prev.next \leftarrow x.next$
2. $x.next.prev \leftarrow x.prev$

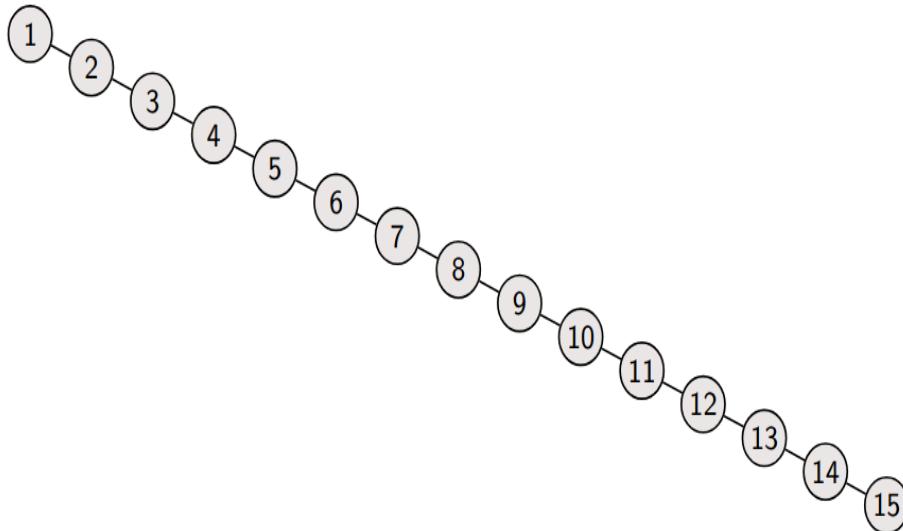
On veut envoyer un flux d'une source à un puits. Le flux est une fonction qui satisfait :

- une contrainte de capacité (pour tout segment edge, $0 \leq f(u,v) \leq c(u,v)$)

- une contrainte de conservation (le flux qui rentre = flux qui sort), $\sum_{\text{for all } v} f(v, u) = \sum_{\text{for all } v} f(u, v)$

valeur d'un flux : flow qui sort de la source - flow qui en revient

Braess Paradox : 4000 people driving from s to t every day. Pour aller le plus vite, il faut que la moitié prenne la route du haut, et l'autre moitié la route du bas.



Ford-Fulkerson

$\Theta(Ef)$

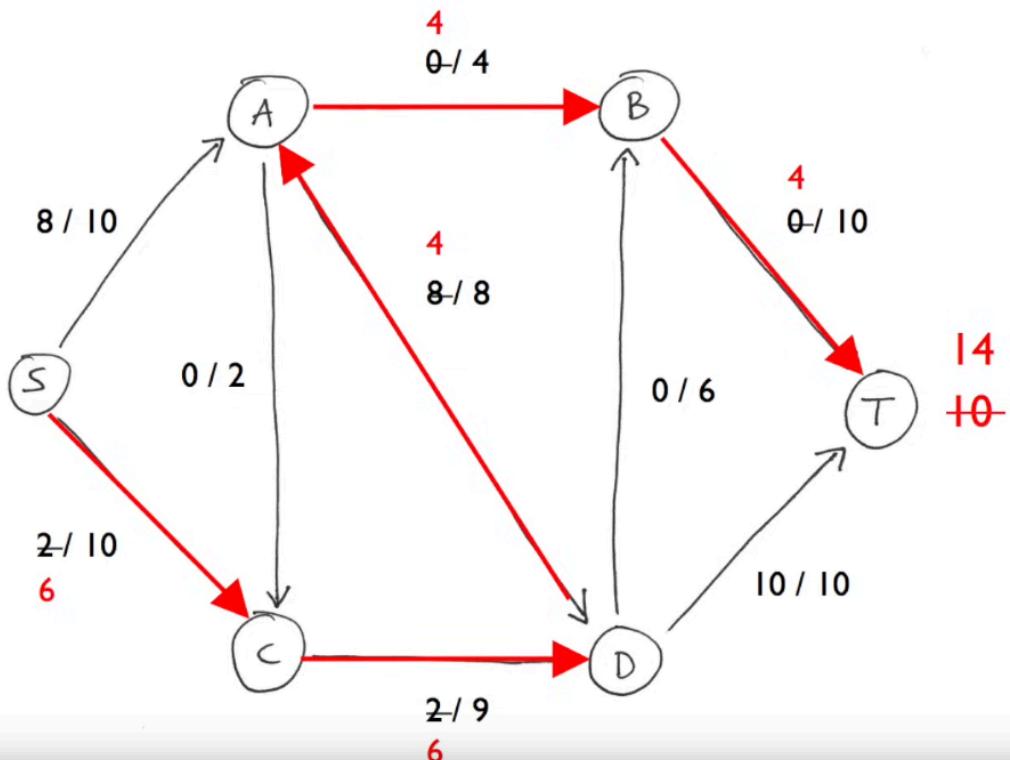
Initialiser les edges à 0 flow.

Répéter jusqu'à ce qu'il n'y ait plus de augmenting path:

1. trouver un augmenting path (avec depth first search), il doit respecter les conditions suivantes :
 1. être un non-full forward edge (un segment qui va vers le puits non déjà remplis)
 2. ou être un non-empty backward edge (on doit pouvoir reverse du flux)
2. calculer la bottleneck capacity (le minimum de capacité d'un edge dans le chemin trouvé)
3. augmenter le total flow out

② Qu'est-ce que ça veut dire "reverse le flux" ?

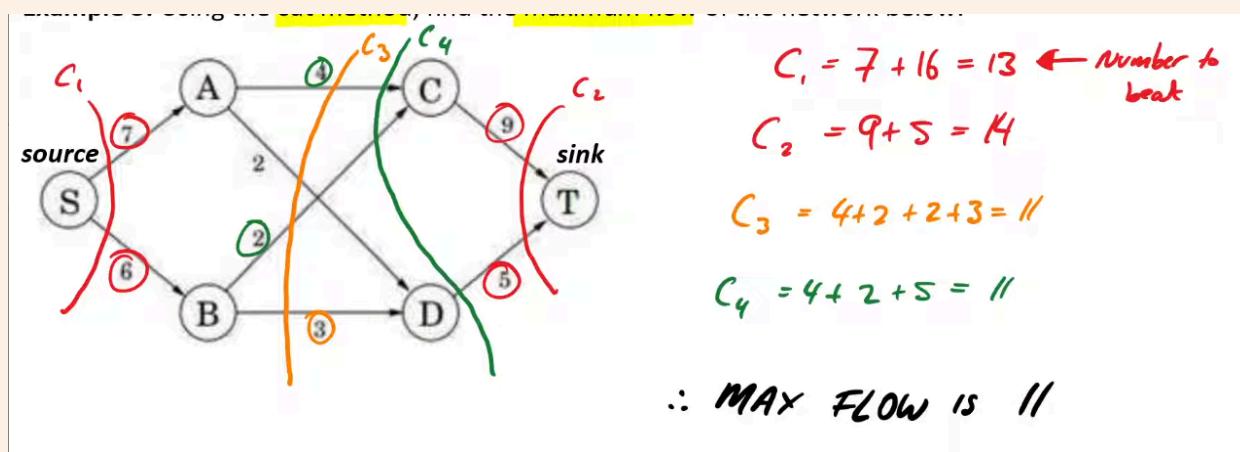
En fait là on voit qu'on a un edge qui va de A vers D avec une capacité de 8. Puis on trouve un chemin qui va dans l'autre sens (et qui pourrait utiliser 4 unités de flux). Le fait qu'on ait 6 qui arrive en bas et plus 2 nous permet d'avoir moins besoin de flux entrant en D, donc on "reverse" on en enlève de A vers D. Et comme on fait ça, on a aussi du flux entrant en A en + qu'on peut réutiliser.



② Min-cut

Si on trouve le "min-cut" d'un flow on trouve aussi le maximum flow!

On compte tous les edges qui vont de avant notre cut jusqu'à après notre cut. On ignore les autres.



Edge-disjoint paths as flow network

- on met toutes les capacités de chaque route à 1
- on calcule le max flow
- The maximum flow value 'f' in this context represents the number of edge disjoint paths between the source and the sink because each path can carry exactly 1 unit of flow, given that each edge's capacity is set to 1. With a maximum flow of 'f', it implies that

there are indeed 'f' paths from the source to the sink that do not share any edges, thus being edge disjoint.

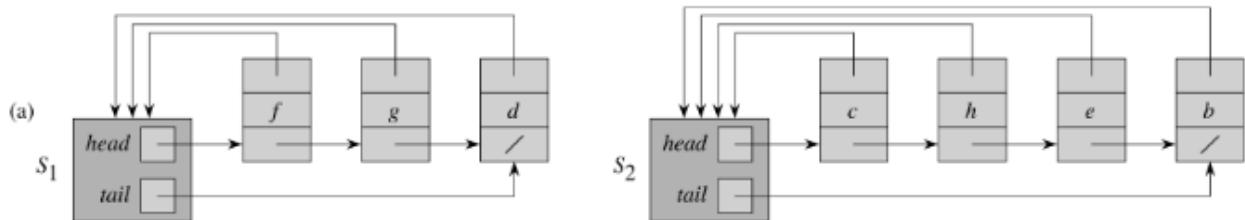
Disjoint-set data structures

On veut un représentant par ensemble pour lui donner un nom.

On ne peut pas avoir deux éléments dans le même ensemble.

- `make-set(x)` : créer un nouveau set $S_i = \{x\}$ et ajouter S_i à S
- `union(x, y)` : si $x \in S_x, y \in S_y$ alors $S = S - S_x - S_y \cup \{S_x \cup S_y\}$
 - détruits S_x et S_y comme ils doivent être disjoints
 - le représentant du nouveau set est n'importe quel membre de $S_x \cup S_y$, souvent le représentant d'un des deux ensembles
- `find(x)` : renvoie le représentant du set dans lequel il y a x

Stocker les sets dans une linked list :



`make`, créer une liste vide en $\Theta(1)$

`union`:

- on append toujours la petite liste à la grande
- on a n_1, n_2 la taille des deux sets. on est en $\Theta(n_2)$. (mais on peut opti en append la liste la courte à l'autre).

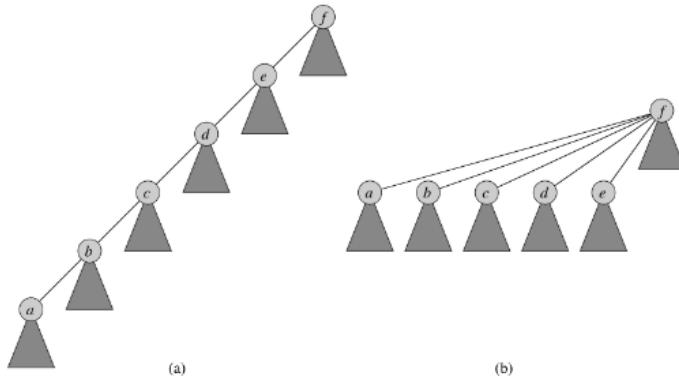
`find`: en $\Theta(1)$ comme chaque noeud pointe vers le représentant

Stocker les sets dans un forest of trees :

- `find`, on remonte les pointeurs jusqu'à la racine, le coût de find dépend de là où est l'élément dans l'arbre

Le rang est une borne supérieure à la hauteur de l'arbre.

Path compression: Find path = nodes visited during FIND on the trip to the root, make all nodes on the find path direct children to root.



On peut optimiser `find` en rendant tous les noeuds un enfant direct du représentant. (on remonte dans l'arbre et on connecte tous les éléments à la racine)

- union by rank : rendre la racine de l'arbre avec le rang le plus petit un enfant direct de la racine de l'arbre plus grand. l'objectif est de garder les arbres les plus horizontaux possibles pour accélérer `find`

Minimum Spanning Trees

Spanning tree : un ensemble T de segments qui touchent tous les noeuds et acyclique.

En entrée : un graph non dirigé $G = (V, E)$ avec des poids $w(u, v)$ pour chaque segment $(u, v) \in E$.

En sortie : un spanning tree avec le coût le plus faible.

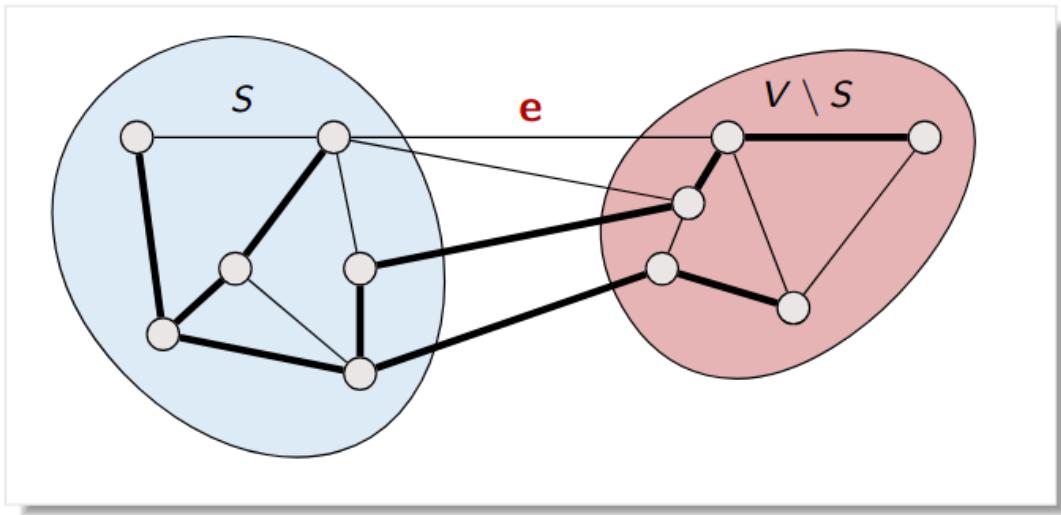
Un **cut** $(S, V \setminus S)$ est une partition des noeuds en deux ensembles non vides disjoints S et $V \setminus S$.

Un **crossing-edge** est un segment qui connecte un noeud dans S à un noeud dans $V \setminus S$.

Si on considère un **cut** $(S, V \setminus S)$ et :

- T est un arbre sur S qui est une partie d'un MST
- e est un crossing-edge de poids minimal

Alors il y a un MST de G qui contient e et T .



Si e est déjà dans le MST, on est bon.

Sinon, on ajoute e . Ça peut créer un cycle. Dans ce cas, il y a au moins un autre crossing edge dans le cycle $w(f) \geq w(e)$. On remplace f par e dans le MST.

On en obtient un nouveau qui contient e et T .

Prim's algorithm

On commence par n'importe quel noeud v et on l'ajoute à T . On a donc un cut induit par T (les noeuds inclus dans T et ceux non).

À chaque étape :

- on ajoute à T un crossing edge de poids minimal par rapport au cut induit par T .

PRIM(G, w, r)

$Q = \emptyset$

for each $u \in G.V$

$u.key = \infty$

$u.\pi = \text{NIL}$

INSERT(Q, u)

DECREASE-KEY($Q, r, 0$) // $r.key = 0$

while $Q \neq \emptyset$

$u = \text{EXTRACT-MIN}(Q)$

for each $v \in G.Adj[u]$

if $v \in Q$ and $w(u, v) < v.key$

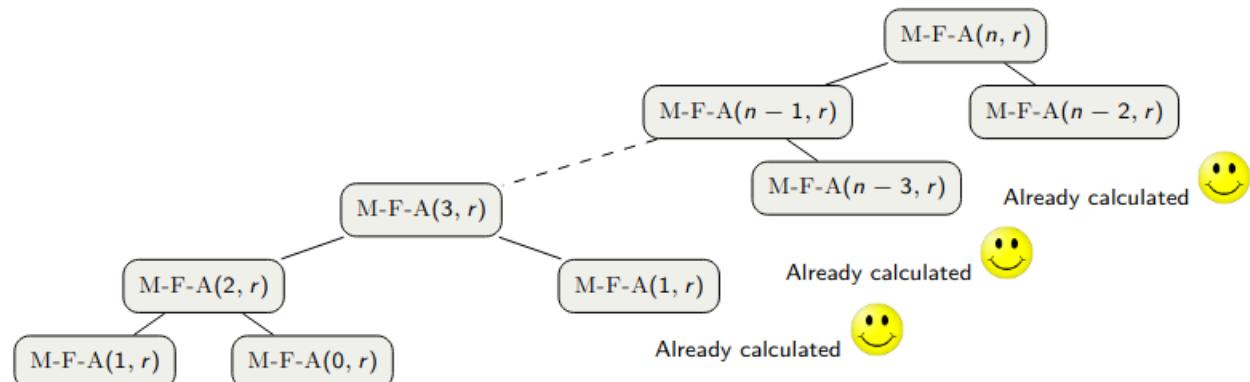
$v.\pi = u$

DECREASE-KEY($Q, v, w(u, v)$)

- π quel est le noeud voisin de u dans T qui est à une distance $u.key$

Au début on dit que chaque vertices est à une distance infinie de T et qu'il n'a pas de voisin. Puis on prend un r au hasard et on dit que sa distance à T est 0. Puis, tant que Q n'est plus vide, on boucle sur tous les vertices, on prend le minimum m .

Pour chaque noeud voisin à ce nouveau minimum, on met à jour sa distance à T (peut-être que le voisin est plus près de m , ou pas, que le noeud précédent).



ecture 9, 18.03.2025

Kruskal's algorithm

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Shortest paths

Entrée :

On autorise les edges avec des valeurs négatives (par exemple des différences d'altitude).

Bellman-Ford algorithm

Loop invariant: après i itérations de la boucle principale ($0 \leq i < |V| - 1$), toute distance $\text{dist}[v]$ correspond à la longueur d'un chemin le plus court composé d'au plus i arêtes.

Dijkstra's algorithm

Si on essaye de lancer Prim's comme d'ha

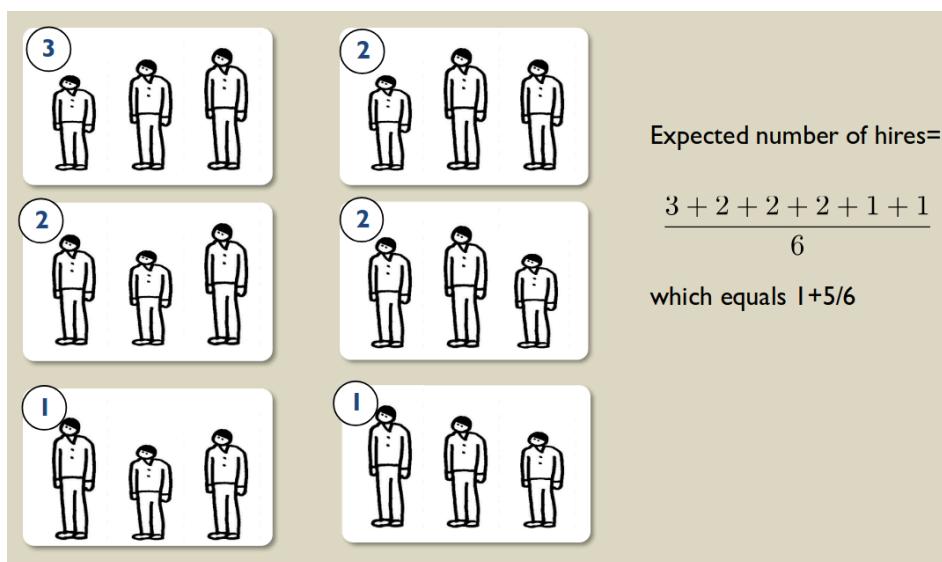
Probabilistic analysis and randomized algorithms

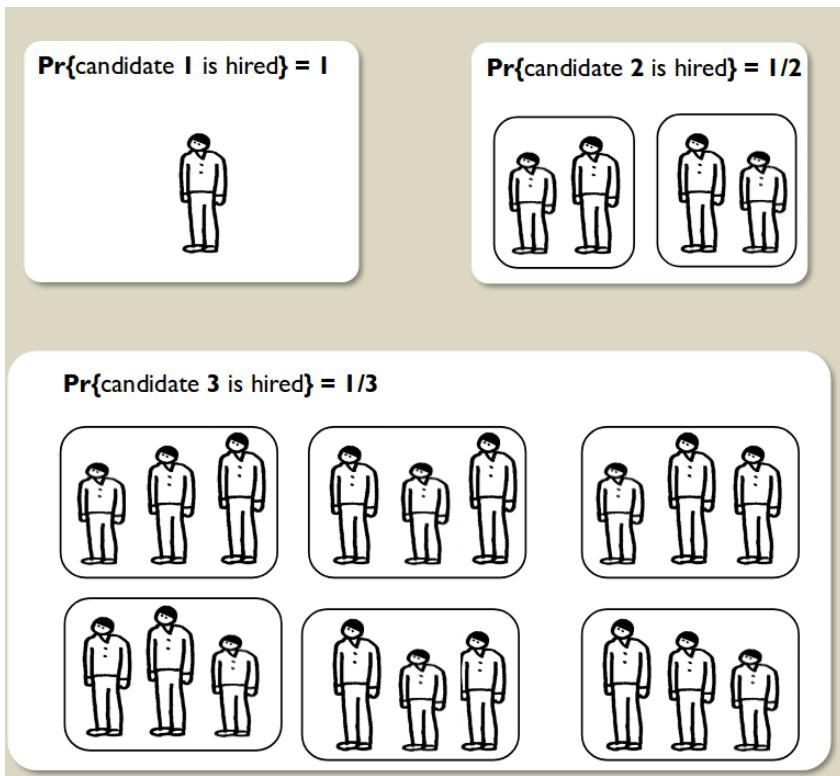
- utile pour ne pas toujours tomber dans le pire cas (et éviter les attaques)

The Hiring problem

On recrute une personne si elle est plus grande que la plus grande personne qu'on a déjà embauché. Le pire cas c'est s'ils arrivent tous en ordre croissant : on va tous les embaucher.

Quel est le nombre d'embauches qu'on va faire parmi toutes les permutations des candidats ?





k smallest numbers in an array

$O(n \log n)$: on trie le tableau, puis on prend les k premiers.

- sélectionner un pivot aléatoire de la liste
- calculer S, L , les ensembles :
 - strictements inférieurs au pivot
 - égaux au pivot
 - strictement supérieurs au pivot
- si $|S| < k$, on sait que tous les éléments qu'on cherche sont dans S
- sinon ils sont dans L

Theory of Computing, Volume 8 (2012), pp. 121–164
www.theoryofcomputing.org

RESEARCH SURVEY

The Multiplicative Weights Update Method:
A Meta-Algorithm and Applications

Sanjeev Arora* Elad Hazan Satyen Kale

Received: July 22, 2008; revised: July 2, 2011; published: May 1, 2012.

Abstract: Algorithms in varied fields use the idea of maintaining a distribution over a certain set and use the *multiplicative update rule* to iteratively change these weights. Their analyses are usually very similar and rely on an exponential potential function.

In this survey we present a simple meta-algorithm that unifies many of these disparate algorithms and derives them as simple instantiations of the meta-algorithm. We feel that since this meta-algorithm and its analysis are so simple, and its applications so broad, it should be a standard part of algorithms courses, like “divide and conquer.”

$\{\text{options or moves}\} = \{m\} = \{1, 2, \dots, m\}$.
costs m_i are unknown and random
(possibly adversarially chosen)

Consider $\gamma \in (0, 1/2)$.
Initialize weights $(w_1, \dots, w_m) = \vec{w}$. } not critical.

Define $\Phi = \sum_{i \in [m]} w_i$.

Choose action $(p_1, \dots, p_m) = \vec{p}$ defined by $\vec{p} = \frac{1}{\Phi} \vec{w}$
(or play option i with probability $p_i = \frac{w_i}{\Phi}$).

Observe costs $(m_1, \dots, m_m) = \vec{m}$ associated to different options

Update weights $w_i \leftarrow w_i (1 - \gamma^{m_i})$. Go back.

Hedge Update of weights: $w_i \leftarrow w_i e^{-\gamma m_i}$

