Algorithmes gloutons

LU3IN003 - Algorithmique
Licence d'Informatique

Chargés de cours : Fanny Pascual, Olivier Spanjaard

Algorithme glouton

Algorithme glouton : Algorithme qui effectue une suite de choix, tels qu'à chaque étape un choix "localement" optimal est effectué et n'est pas remis en question par la suite.

Exemples :

- Algorithme de Dijkstra : à chaque étape le sommet ouvert le plus proche de la racine est ajouté à l'arborescence courante.

- Algorithme de Prim : à chaque étape le sommet ouvert le plus proche de l'arbre courant est ajouté à l'arbre.

Algorithme glouton optimal

Pour certains problèmes, il existe des algorithmes gloutons qui retournent des solutions optimales. On a alors les propriétés :

Propriété de choix glouton : il existe toujours une solution optimale qui contient un premier choix glouton.
$\rightarrow$ On peut toujours arriver à une solution optimale en faisant un choix localement optimal.

Propriété de sous-structure optimale : trouver une solution optimale contenant le premier choix glouton se réduit à trouver une solution optimale pour un sous-problème de même nature.

Suite du cours : exemples

- Arbre couvrant de coût minimum : algorithme de Kruskal

- Compression de textes : algorithme de Huffman

- Ordonnancement d'intervalles : algorithme glouton

Algorithme de Kruskal

Joseph Kruskal (1928 - 2010)

Mathématicien ,
chercheur en informatique,
et psychométricien.

Travaille aux laboratoires Bell.

Algorithme de Kruskal

Principe de l'algorithme :
(H est un arbre couvrant de coût minimum à la terminaison de l'algorithme)

Algorithme de Kruskal

Trier les arêtes par coût croissant;

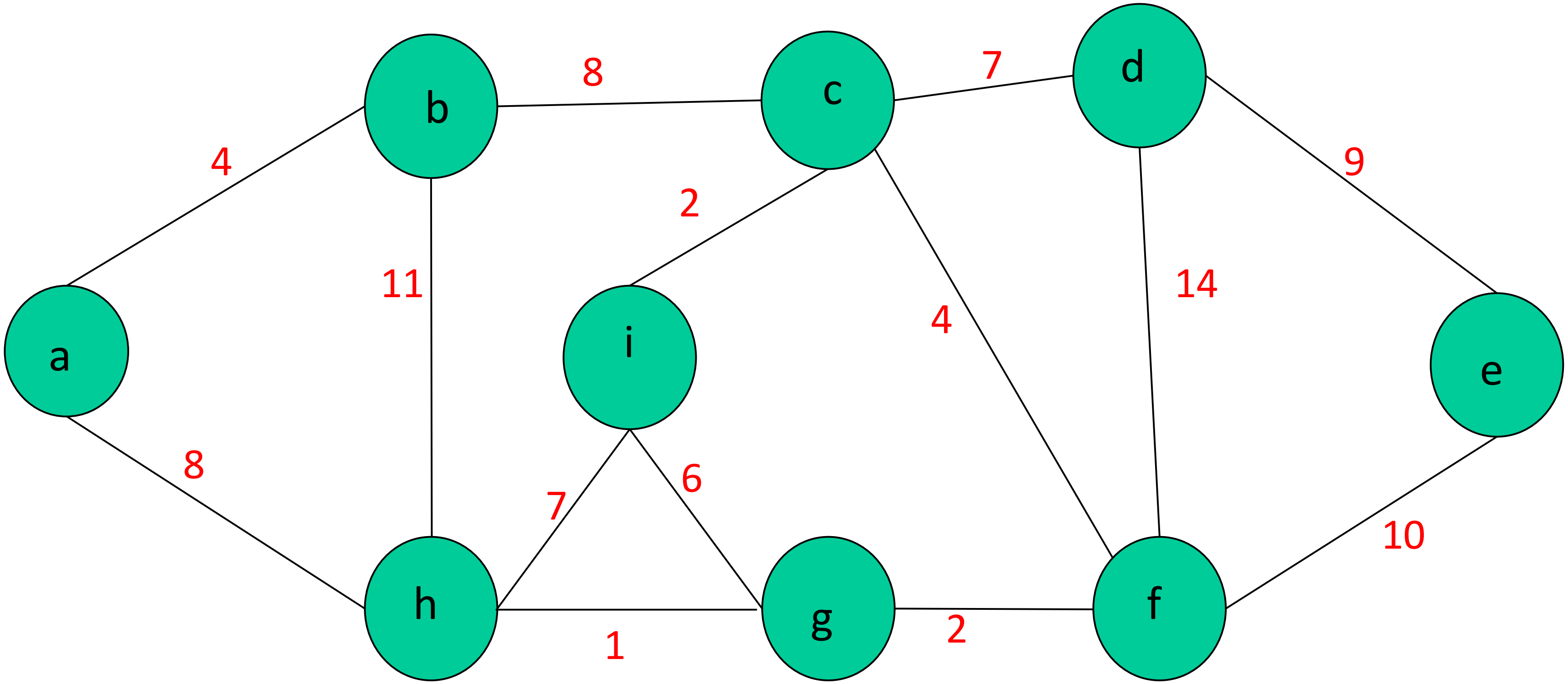
H = arbre vide;

Examiner dans l'ordre chacune des arêtes {x,y} :

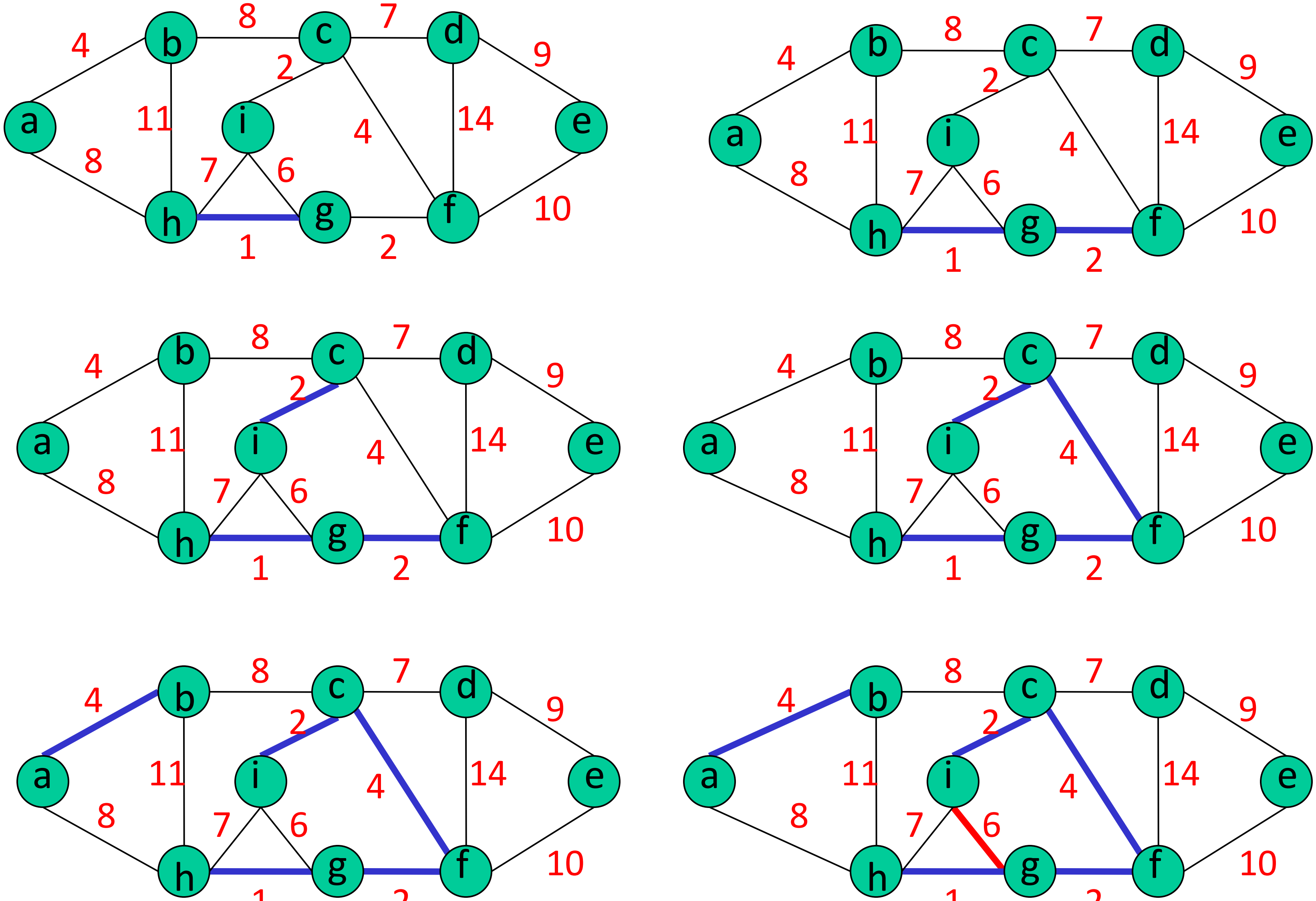
S'il n'existe pas de chaîne de x à y dans H :

H = H U {x,y}

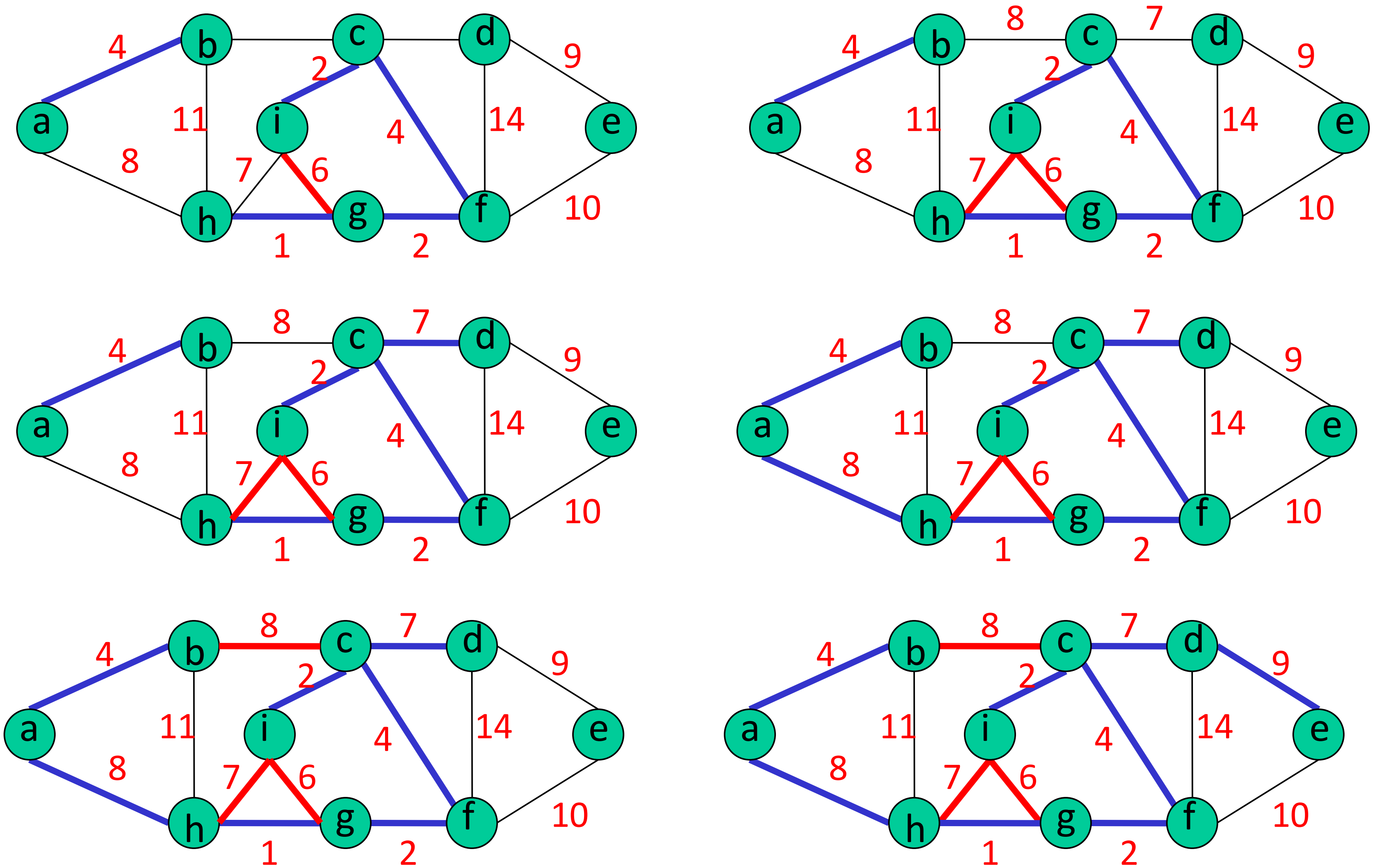
FinSi


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
10
7
6
1
2

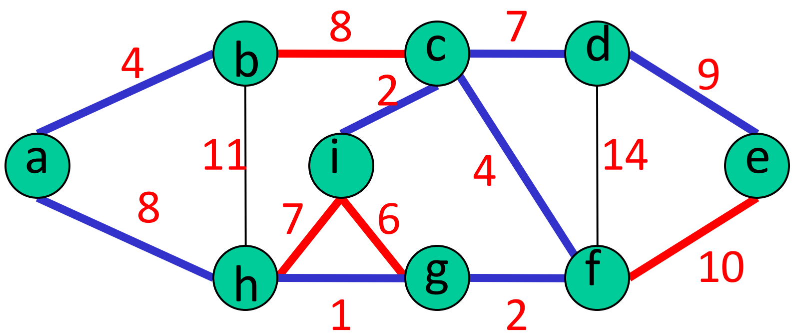
Une exécution de l'algorithme de Kruskal

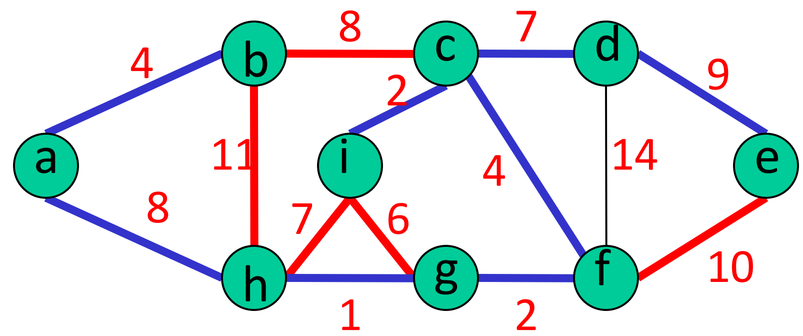


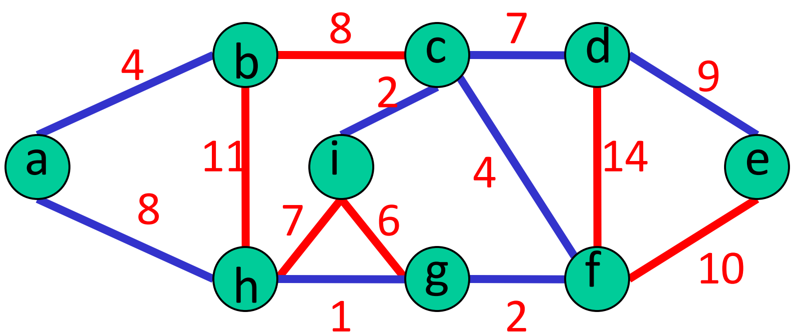
Une exécution de l'algorithme de Kruskal (suite)



Une exécution de l'algorithme de Kruskal (fin)







Quiz

Quelle est la complexité de l’algorithme de Kruskal ?

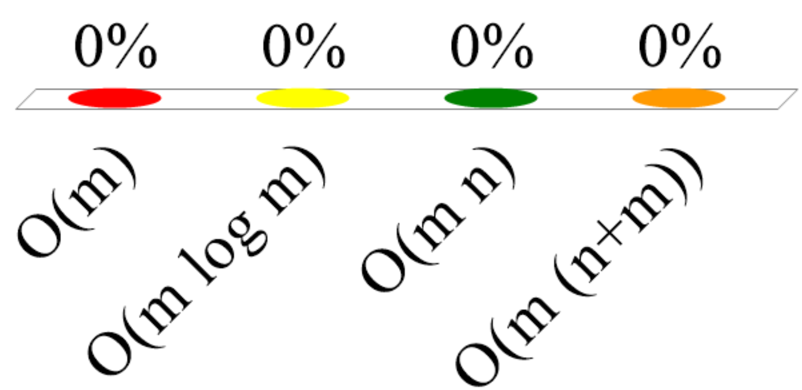
```
Algorithme de Kruskal:
   Trier les arêtes par coût croissant;
   H = arbre vide;
   Examiner l’arête {x,y} :
   s’il n’existe pas de chaîne de x à y dans H
         H = H U {x,y}
   FinSi
```

A. O(m)

B. O(m log m)

C. O(m n)

D. O(m (n+m))



Complexité de l’algorithme de Kruskal

Algorithme de Kruskal

Trier les arêtes par coût croissant;

H = arbre vide;

Examiner dans l’ordre chacune des arêtes {x,y} :

S’il n’existe pas de chaîne de x à y dans H :

H = H U {x,y}

FinSi

Complexité de l'algorithme de Kruskal

Algorithme de Kruskal

Trier les arêtes par coût croissant;

H = arbre vide;

Examiner dans l'ordre chacune des arêtes {x,y} :

S'il n'existe pas de chaîne de x à y dans H :

H = H U {x,y}

FinSi

Trier (ensemble des arêtes) se fait en O(m log m)

Soit α la complexité de la comparaison

« Composante connexe (x) = composante connexe (y) »

La complexité de l'algorithme est $O(m \log m + m\,\alpha)$

Montrons que la complexité est en $O(m \log m)$

Union-Find (unir – trouver)

On souhaite travailler sur les partitions de E={1, …, n}

Exemple :

Partition = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }

classe d'équivalence 1 2 3 4

Etant donné une partition, on souhaite

- Trouver la classe d'équivalence d'un élément : méthode Find
- Fusionner deux classes d'équivalence : méthode Union

Dans l'algorithme de Kruskal : E = ensemble des sommets.

- Deux sommets x et y sont dans une même composante connexe si Find(x)= Find(y)
- Si on choisit l'arête {x,y}, on fusionne deux composantes connexes en une : Union(x,y)

Algorithme de Kruskal avec Union-Find

```
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l'ordre chacune des arêtes {x,y} :
  s'il n'existe pas de chaîne de x à y dans H
     H = H U {x,y}
  FinSi
```

```
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l'ordre chacune des arêtes {x,y} :
  si Find(x) ≠ Find(y)
     H = H U {x,y}
     Union(x,y)
  FinSi
```

Une solution peu efficace

On représente la partition par un tableau tab tel que tab[i] est le numéro de la classe d'équivalence de l'élément i.

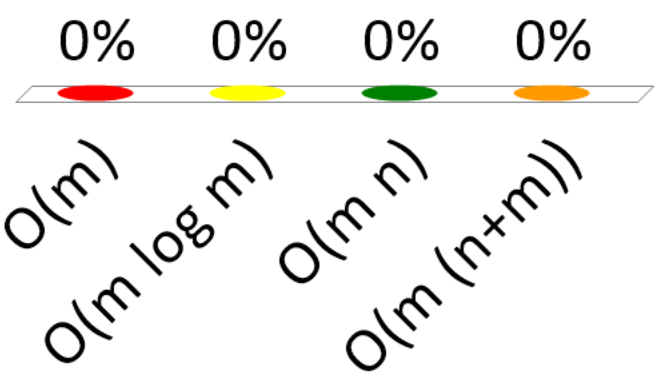
P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }
classe 1 2 3 4

1	2	3	4	5	6	7	8	9	10
1	**3**	**1**	**2**	**4**	**4**	**3**	**4**	**3**	**4**

Quiz Quelle est la complexité de l'algorithme de Kruskal si on code Union-Find avec ce tableau ?

Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l'ordre chacune des arêtes {x,y}
  si Find(x) ≠ Find(y)
    H = H U {x,y}; Union(x,y)
  FinSi

A. O(m)
B. O(m log m)
C. O(m n)
D. O(m (n+m))



Une solution peu efficace

On représente la partition par un tableau tab tel que tab[i] est le numéro de la classe d'équivalence de l'élément i.

P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }
classe 1 2 3 4

1	2	3	4	5	6	7	8	9	10
1	**3**	**1**	**2**	**4**	**4**	**3**	**4**	**3**	**4**

Find : complexité en O(1)

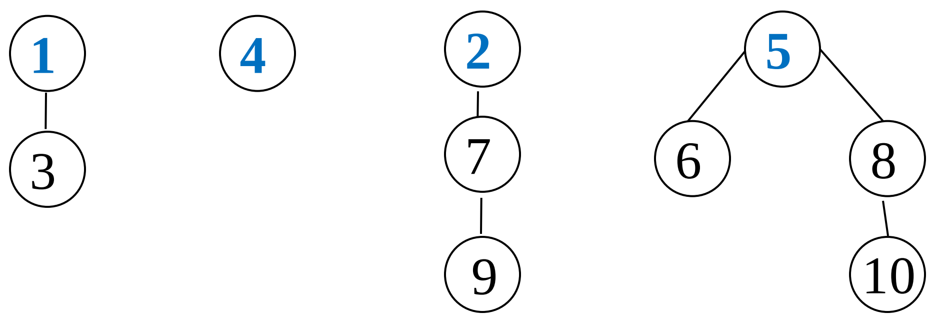
Union : complexité en O(n)

Une meilleure solution

On représente la partition par une forêt.
- Chaque arbre correspond à une classe d'équivalence.
- La racine de chaque arbre est le "représentant" de la classe.

On représente la forêt par un tableau père, tel que père[i] est le père de l'élément i, en posant père[i]=i pour une racine.

P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }

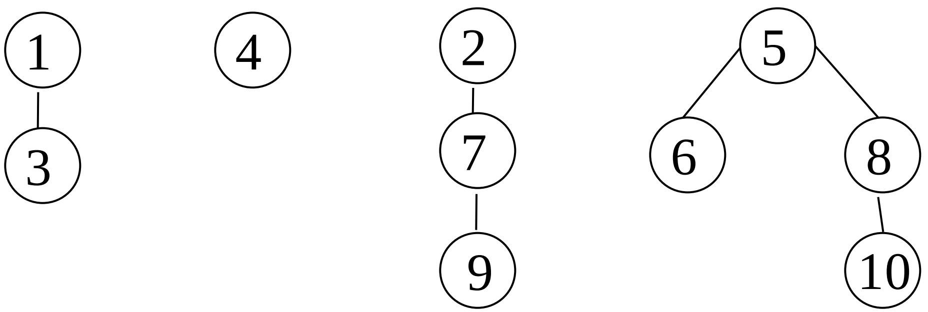


	1	2	3	4	5	6	7	8	9	10
père	**1**	**2**	**1**	**4**	**5**	**5**	**2**	**5**	**7**	**8**

Find

```
fonction Find(entier x): entier;
Tant que (x ≠ père[x])
        x := père[x];
FinTantque
Retourner(x)
```

Exemple: Find(9)   P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



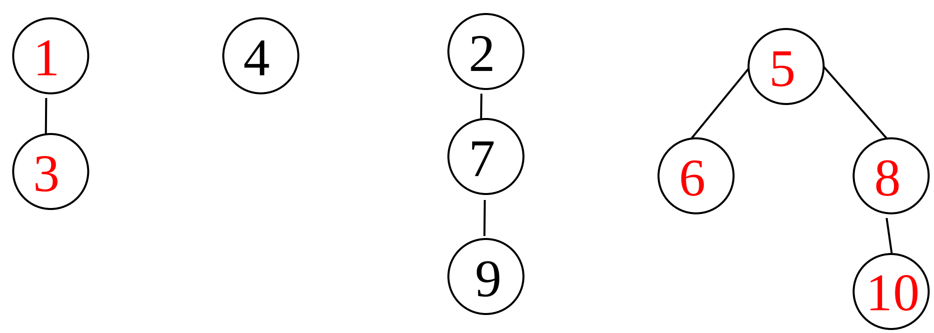
	1	2	3	4	5	6	7	8	9	10
père	**1**	**2**	**1**	**4**	**5**	**5**	**2**	**5**	**7**	**8**

père[9]=7 ; père[7]=2; père[2]=2; retourner(2)

Union

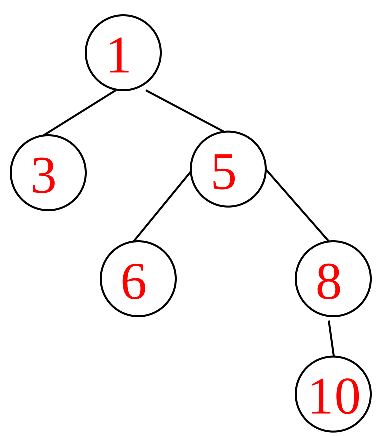
```
procédure Union (entier x, entier y): entier;
r1= Find(x);
r2 = Find(y);
Si (r1  ≠ r2) alors
            père[r2] = r1;
FinSi
```

Exemple: Union(3,8) P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



Find(3)=1
Find(8)=5

père[5]=1

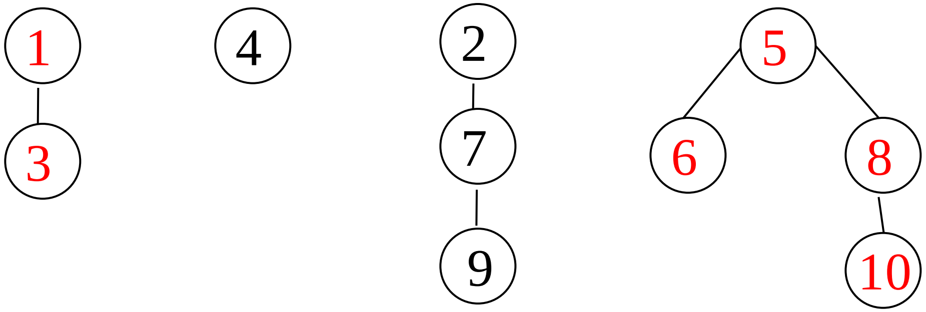


L'opération Find s'effectue en temps O(h), où h est la hauteur de l'arbre. Pire cas : h= O(n)

Union pondérée

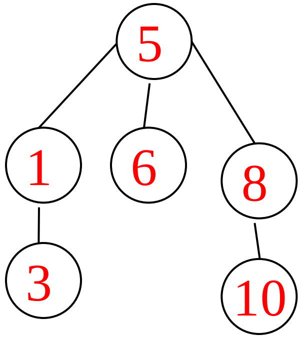
Lors de l'union de deux arbres, la racine de l'arbre avec le moins de sommets devient fils de la racine de l'autre.

Exemple: Union(3,8) P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



Find(3)=1
Find(8)=5

père[1]=5



Union pondérée

On maintient un tableau taille (initialisé à 1 si l’on part de classes singleton) : si i est une racine, taille[i] est le nombre de sommets de l’arbre de racine i.

```
procédure Union (entier x, entier y): entier;
r1= Find(x);
r2 = Find(y);
Si (r1 ≠ r2) alors
        Si (taille[r1])>taille[r2]) alors
                père[r2] := r1;
                taille[r1] :=taille[r1]+taille[r2]
        sinon
                père[r1] := r2;
                taille[r2] :=taille[r1]+taille[r2]
        FinSi
FinSi
```

Union pondérée

La hauteur d'un arbre à n sommets créé par une suite d'unions pondérées est $\leq 1 + \lfloor \log_2(n) \rfloor$

Preuve: Par récurrence sur n

- *Cas de base :* vrai pour n=1.

-*Induction :*

Soit T un arbre obtenu par union pondérée d'un arbre à x sommets (avec $1 \leq x \leq n/2$) et d'un arbre à (n-x) sommets.
Hauteur(T) $\leq \max(1 + \lfloor \log_2(n-x) \rfloor, 2 + \lfloor \log_2(x) \rfloor)$.

Or $\log_2(n-x) \leq \log_2(n)$
et $\log_2(x) \leq \log_2(n/2) \leq \log_2(n)-1$
La hauteur de T est donc majorée par $1 + \lfloor \log_2(n) \rfloor$

Union et Find sont donc en O(log n)

Point sur la complexité de l'algorithme de Kruskal

```
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l'ordre chacune des arêtes {x,y} :
   si Find(x) ≠ Find(y)
         H = H U {x,y}
         Union(x,y)
   FinSi
```

Union et Find sont en $O(\log n)$

Complexité de Kruskal : $O(m \log m + m \log n) = O(m \log m)$

Et si les coûts des arêtes sont faibles (en $O(m)$), de façon à ce que Trier(ensemble des arêtes) se fasse en $O(m)$?

On cherche à améliorer la complexité de Union et Find.

Deuxième amélioration : compresser les chemins

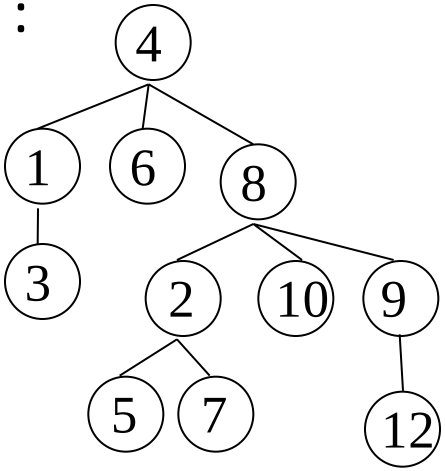
Quand on remonte du sommet x vers sa racine r, on refait le parcours en faisant de chaque sommet rencontré un fils de r.

```
fonction Findsimple(entier x):
entier;
Tant que (x ≠père[x])
          x := père[x];
FinTantque
Retourner(x)
```

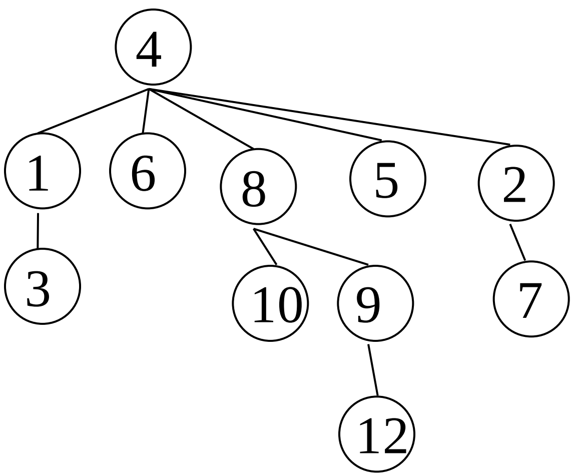
```
fonction Find(entier x): entier;
r :=  Findsimple(x);
Tant que (x ≠ père[x])
          y := père[x]
          père[x]:=r;
          x:=y;
FinTantque
Retourner(x)
```

Exemple : Find(5)

Arbre initial :



Arbre final :



fonction Find(entier x): entier;
r := Findsimple(x);
Tant que (x ≠ père[x])
    y := père[x]
    père[x]:=r;
    x:=y;
FinTantque
Retourner(x)

Complexité de Union et Find ?

Complexité amortie : n opérations Union + m opérations Find se réalisent en temps $O(n + m\,\alpha(n,m))$, où α est une fonction qui croît très très lentement.

En effet, $\alpha(n,m)$ est la réciproque de la fonction d'Ackermann qui croît extrêmement vite: on a $\alpha(n,m) \leq 4$ pour $n,m \leq 2^{2048}$.
En pratique, $\alpha(n,m)$ est donc une constante.

Complexité de l'algorithme de Kruskal ?

Si on suppose que Tri(arêtes) est effectué en temps O(m).

On a alors n-1 operations Union et m operations Find pour n éléments : ceci se fait en $O(n + m\,\alpha(n,m))$.

Rappel: dans le cas général la complexité est en O(m log m).

Quiz

Soit G un graphe connexe. Quel algorithme choisir pour trouver un arbre couvrant de coût minimum de G si G est peu dense (m = 10 n) ?

A. Algo. de Prim sans tas

B. Algo de Prim avec tas

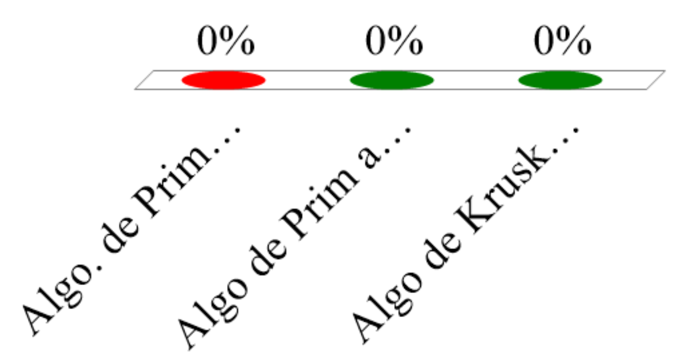
C. Algo de Kruskal avec union pondérée et compression de chemins

- Prim sans tas : $O(n^2)$

- Prim avec tas : $O(m \log n)$

- Kruskal :

- cas général : $O(m \log m)$
- coûts des arêtes en $O(n+m)$: $O(n + m\,\alpha(n,m))$.



Quiz

Quel algorithme choisir pour trouver un arbre couvrant de coût minimum de G si G est très dense ($m = 0.2\ n^2$) ?

A. Algo. de Prim sans tas

B. Algo de Prim avec tas

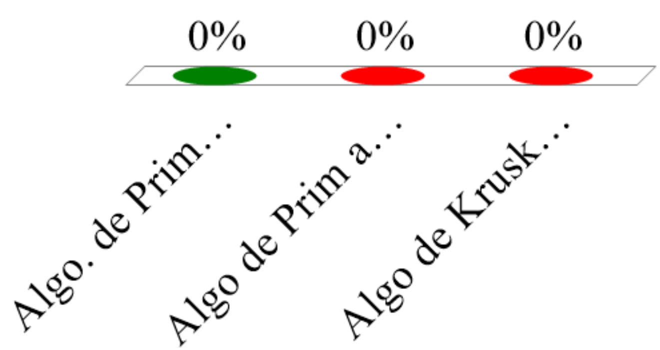
C. Algo de Kruskal avec union pondérée et compression de chemins

- Prim sans tas : $O(n^2)$

- Prim avec tas : $O(m \log n)$

- Kruskal :

- cas général : $O(m \log m)$
- coûts des arêtes en $O(n+m)$: $O(n + m\,\alpha(n,m))$.



Quiz

Quel algorithme choisir pour trouver un arbre couvrant de coût minimum de G si G est peu dense et les coûts des arêtes sont inférieurs à 10m ?

A. Algo. de Prim sans tas

B. Algo de Prim avec tas

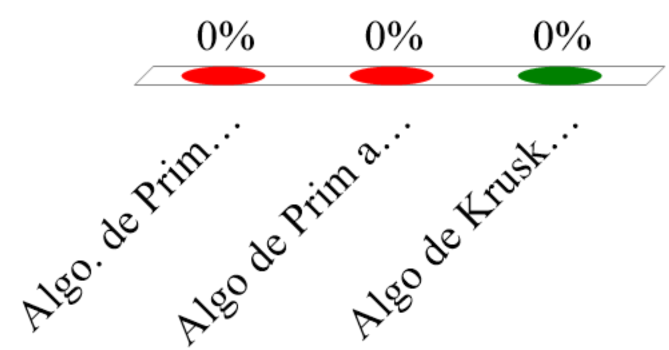
C. Algo de Kruskal avec union pondérée et compression de chemins

- Prim sans tas : $O(n^2)$

- Prim avec tas : $O(m \log n)$

- Kruskal :

- cas général : $O(m \log m)$
- coûts des arêtes en $O(n+m)$: $O(n + m\, \alpha(n,m))$.



Ordonnancement d’intervalles

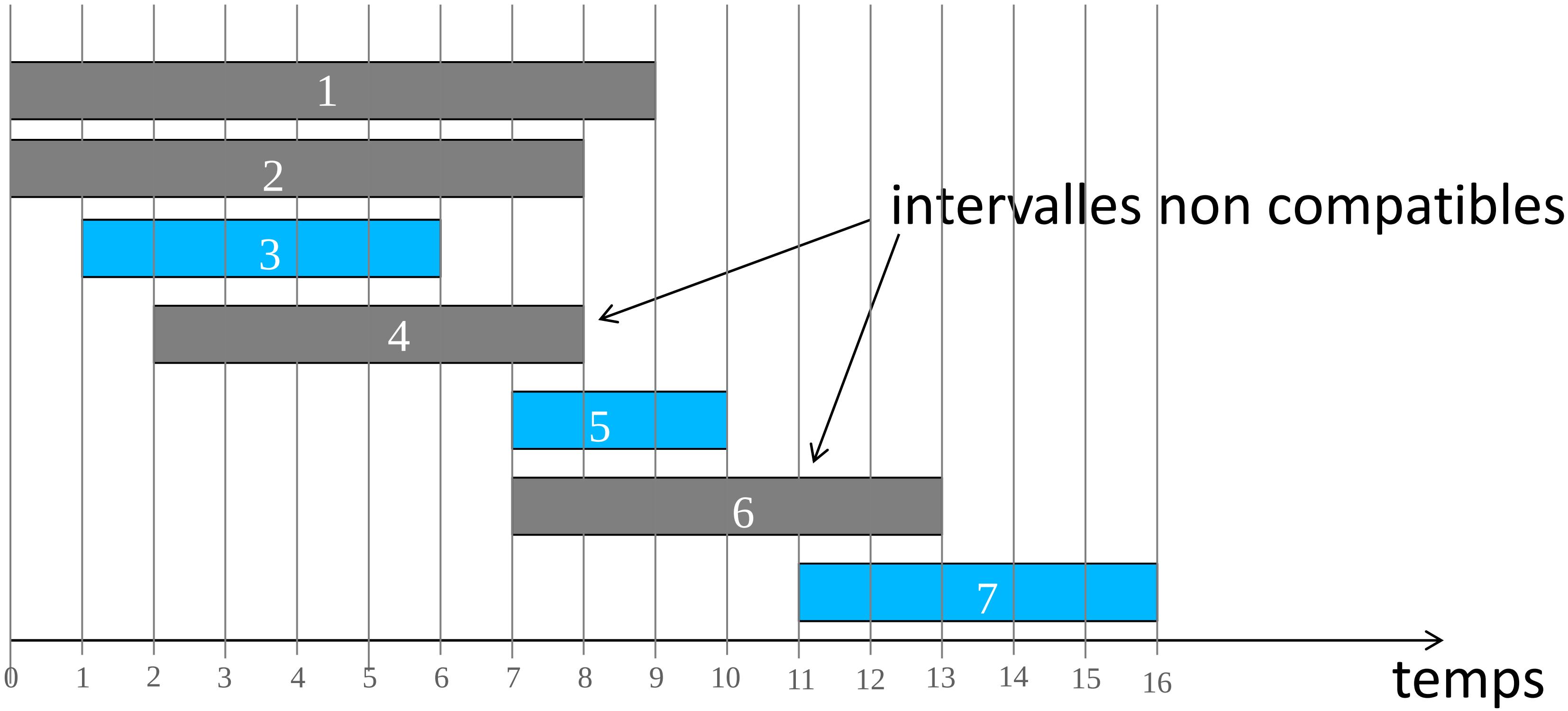
Application : Réservation de salle.

Une salle municipale peut être réservée par diverses associations. Chaque association indique un intervalle durant lequel elle souhaite disposer de la salle.

But : maximiser le nombre d’associations satisfaites.

Ordonnancement d’intervalles

- L’intervalle i commence en d_i et se termine en f_i.
- Deux intervalles sont compatibles s’ils ne s’intersectent pas.
- But : déterminer un sous-ensemble d’intervalles mutuellement compatibles de taille maximale.



Algorithmes gloutons

Algorithme générique :

- Examiner les intervalles dans un ordre spécifique.
- Prendre chaque intervalle dans cet ordre s'il est compatible avec les intervalles déjà pris.

- L'intervalle le plus court d'abord : on considère les intervalles par ordre de $(f_i - d_i)$ croissant.

- L'intervalle qui a le moins de conflits d'abord : pour chaque intervalle i, soit c_i le nombre d'intervalles avec lesquels il est en conflit ; on considère les intervalles par ordre de c_i croissant.

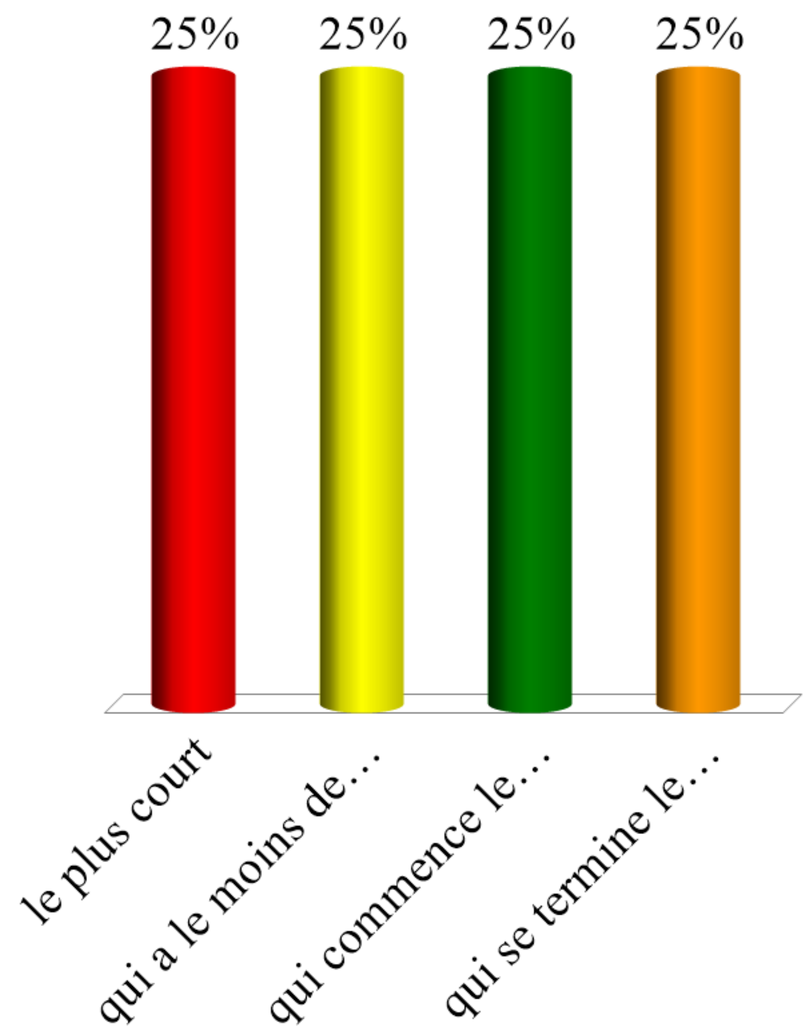
- L'intervalle qui commence le plus tôt d'abord : on considère les intervalles par ordre de d_i croissant.

- l'intervalle qui se termine le plus tôt d'abord : on considère les intervalles par ordre de f_i croissant.

Quiz Quel algorithme glouton est optimal ?

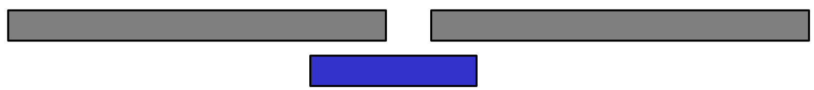
Il s'agit de l'algorithme glouton qui sélectionne d'abord l'intervalle …

A. le plus court
B. qui a le moins de conflits
C. qui commence le plus tôt
D. qui se termine le plus tôt

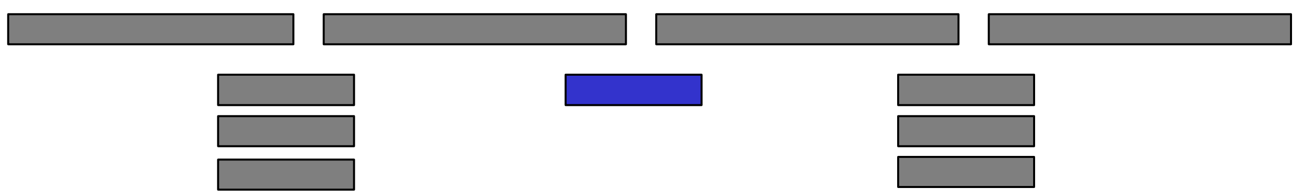


Algorithmes gloutons

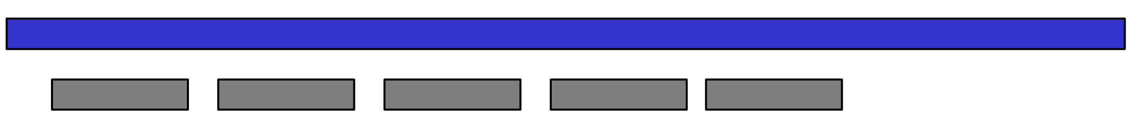
- L’intervalle le plus court d’abord : contre-exemple



- L’intervalle qui a le moins de conflits d’abord : contre-exemple



- L’intervalle qui commence le plus tôt d’abord : contre-exemple



Celui qui se termine le plus tôt d'abord

PlusPetiteDateDeFinDabord $(n, d_1, d_2, \ldots, d_n, f_1, f_2, \ldots, f_n)$

Trier les intervalles par dates de fin t.q. $f_1 \leq f_2 \leq \ldots \leq f_n$
IntervallesChoisis := $\varnothing$

Pour i allant de 1 à n
    Si i est compatible avec IntervallesChoisis
        IntervallesChoisis := IntervallesChoisis $\cup$ {i}

Retourner IntervallesChoisis

Propriétés :

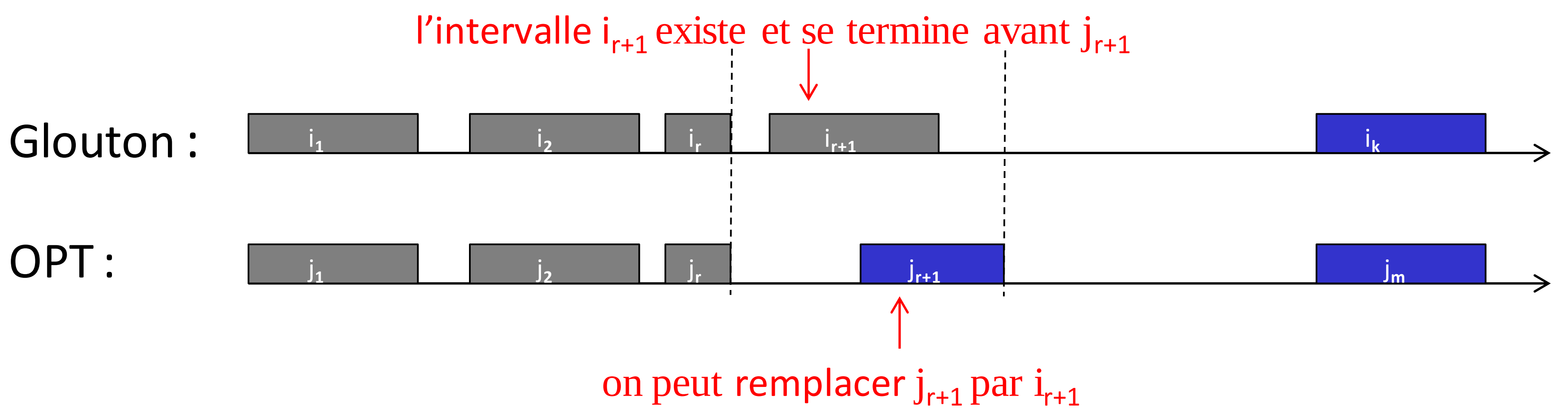
- On garde en mémoire l'intervalle k qui a été ajouté en dernier à IntervallesChoisis.
- L'intervalle j est compatible avec IntervallesChoisis ssi $d_j \geq f_k$
- Algorithme en O(n log n) (complexité du tri)

Analyse de l'algorithme

Théorème : L'algorithme qui considère les intervalles par ordre des f_i croissant est optimal.

Preuve : par l'absurde.

- Supposons que cet algorithme ne soit pas optimal.
- Soit $i_1, i_2, \dots, i_k$ les intervalles sélectionnés par cet algorithme.
- Soit $j_1, j_2, \dots, j_m$ les intervalles sélectionnés par un algorithme optimal avec $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$ avec r le plus grand possible.

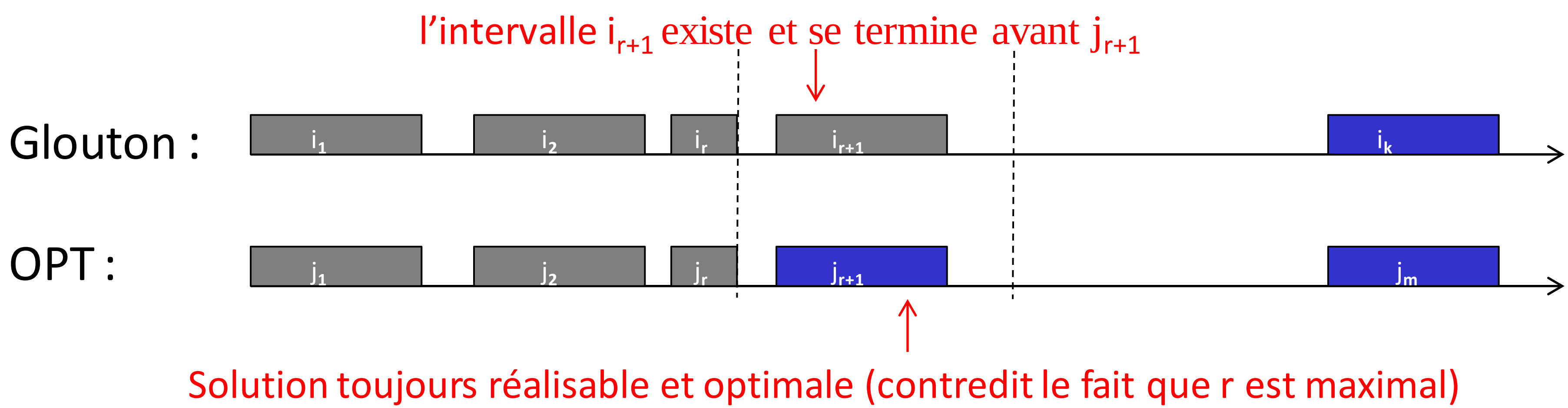


Analyse de l'algorithme

Théorème : L'algorithme qui considère les intervalles par ordre des f_i croissant est optimal.

Preuve : par l'absurde.

- Supposons que cet algorithme ne soit pas optimal.
- Soit $i_1, i_2, \ldots, i_k$ les intervalles sélectionnés par cet algorithme.
- Soit $j_1, j_2, \ldots, j_m$ les intervalles sélectionnés par un algorithme optimal avec $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ avec r le plus grand possible.



Suite du cours

- Arbre couvrant de coût minimum : algorithme de Kruskal

- Ordonnancement d’intervalles : algorithme glouton

- Compression de textes : algorithme de Huffman

Codage de Huffman

Données : une chaîne de caractères (= un fichier)

On souhaite encoder le fichier en un nombre minimum de bits (= compresser le fichier)

Codage par symbole : chaque caractère est remplacé par un code (ensemble de bits) correspondant.

- Codage à longueur fixe : codes de même longueur
- Codage à longueur variable : les codes peuvent être de longueurs différentes.

→ codage de Huffman

Codage de Huffman

Algorithme de codage inventé par David Albert Huffman lors de sa thèse au MIT en 1952.

C’est le meilleur algorithme de codage par symbole possible.

David Albert Huffman (1925-1999)

Principe du codage de Huffman : utiliser des codes courts pour les caractères fréquents et plus longs pour les caractères les moins fréquents

caractère	a	b	c	d	e	f	nombre de bits
fréquence	45	13	12	16	9	5	
longueur fixe	000	001	010	011	100	101	300
longueur variable	0	101	100	111	1101	1100	224

Comment décoder ?

Avec un code à longueur fixe (x), c'est facile, on découpe le texte en sous-chaînes de longueur x.

	fixe	variable
a	000	0
b	001	101
c	010	100
d	011	111
e	111	1101

Exemple : 0 0 0 | 0 0 1 | 0 0 0 | 1 1 1

a b a e

Avec un code à longueur variable, il faut s'assurer que le code d'un caractère n'est pas le préfixe d'un autre !

→ un code est appelé code préfixe si il vérifie cette propriété.

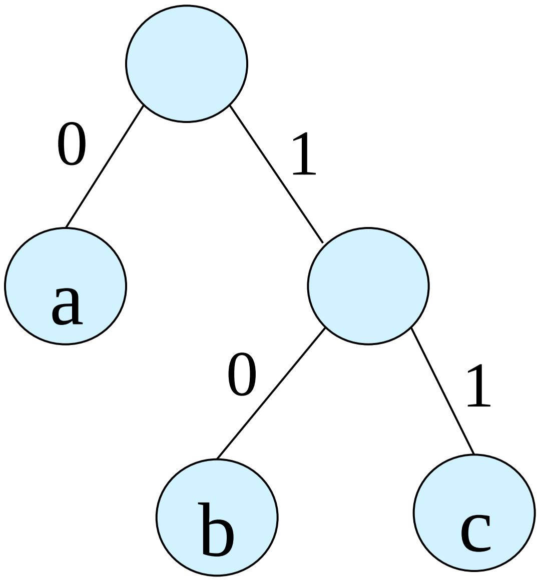
On le décode alors de façon non ambigüe.

Exemple : 0 | 1 0 1 | 0 | 1 1 0 1

a b a e

Comment créer un code préfixe ?

On considère un arbre binaire, avec :
- 0 signifiant descendre à gauche,
- 1 signifiant descendre à droite.



Le code d'un caractère est le code du chemin de la racine à ce caractère.

Exemple :

Caractère	code
a	0
b	10
c	11

C'est un code préfixe car chaque feuille a un unique chemin qui y mène et ne se prolonge pas.

Code préfixe optimal

Etant donnée une chaîne de caractères sur un alphabet $A=\{ a_1 , \ldots , a_n \}$, avec des fréquences $f(a_i)$, on veut déterminer un code préfixe qui minimise le nombre total de bits :

$$\sum_{a=1}^{n} f(a_i) L(c(a_i))$$

où - $c(a_i)$ est le code de a_i

- $L(c(a_i))$ le nombre de bits dans $c(a_i)$.
  ($L(c(a_i))=\mathrm{prof}(a_i)$, la profondeur de a_i dans l'arbre de codage).

Un code préfixe optimal (codage de Huffman) peut être déterminé à l'aide d'un algorithme glouton.

Algorithme glouton

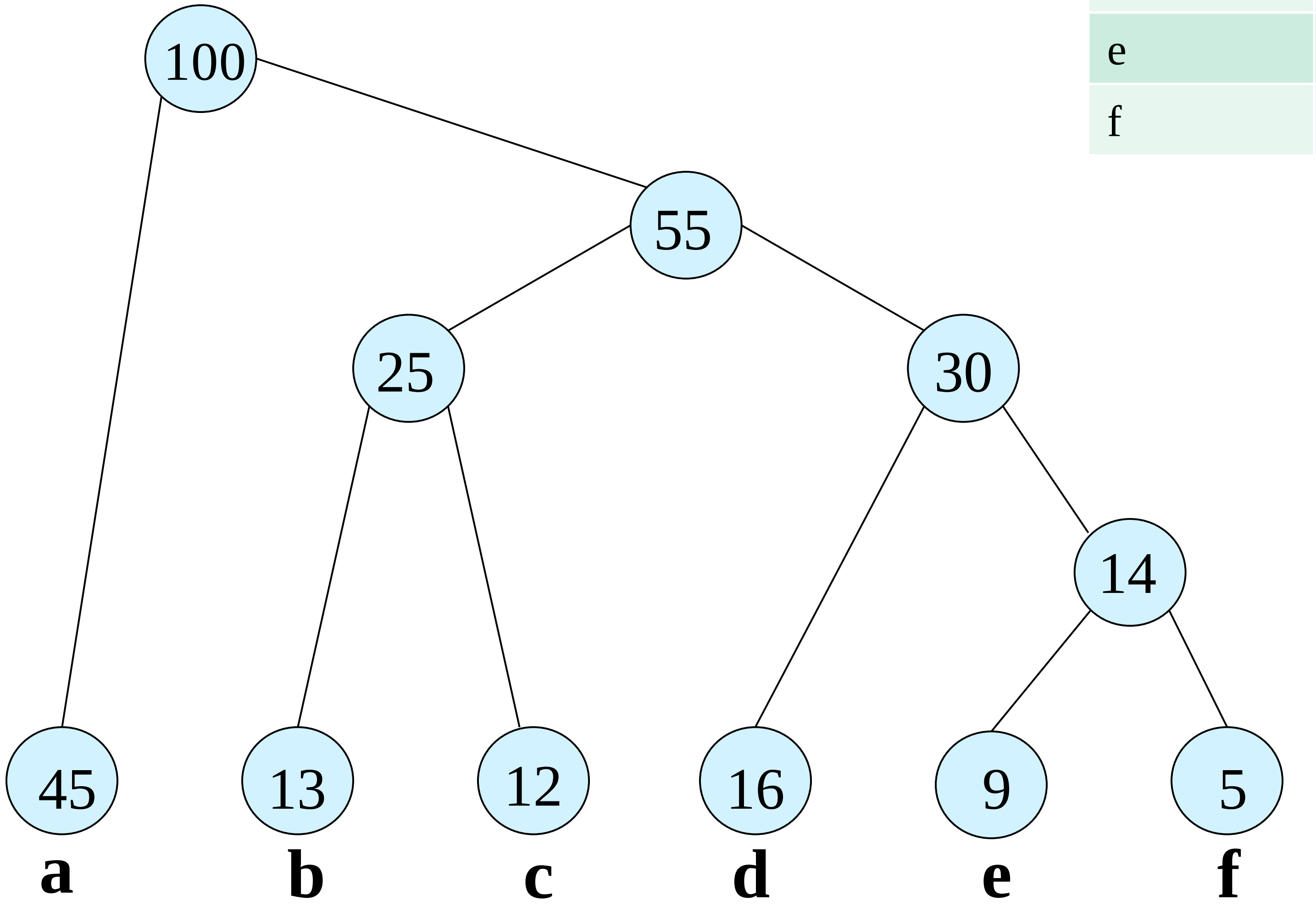
Les feuilles de l'arbre sont les caractères, associés à leur fréquence, et les nœuds internes contiennent la fréquence des caractères du sous arbre.

Algorithme :

- Pour chaque caractère a_i créer une feuille de poids $f(a_i)$
- Tant qu'il existe plusieurs arbres :
  - On choisi deux arbres T_1 et T_2 de poids minimal.
  - On créé un arbre de racine poids(T_1) + poids (T_2) et ayant comme sous-arbre T_1 et T_2 .

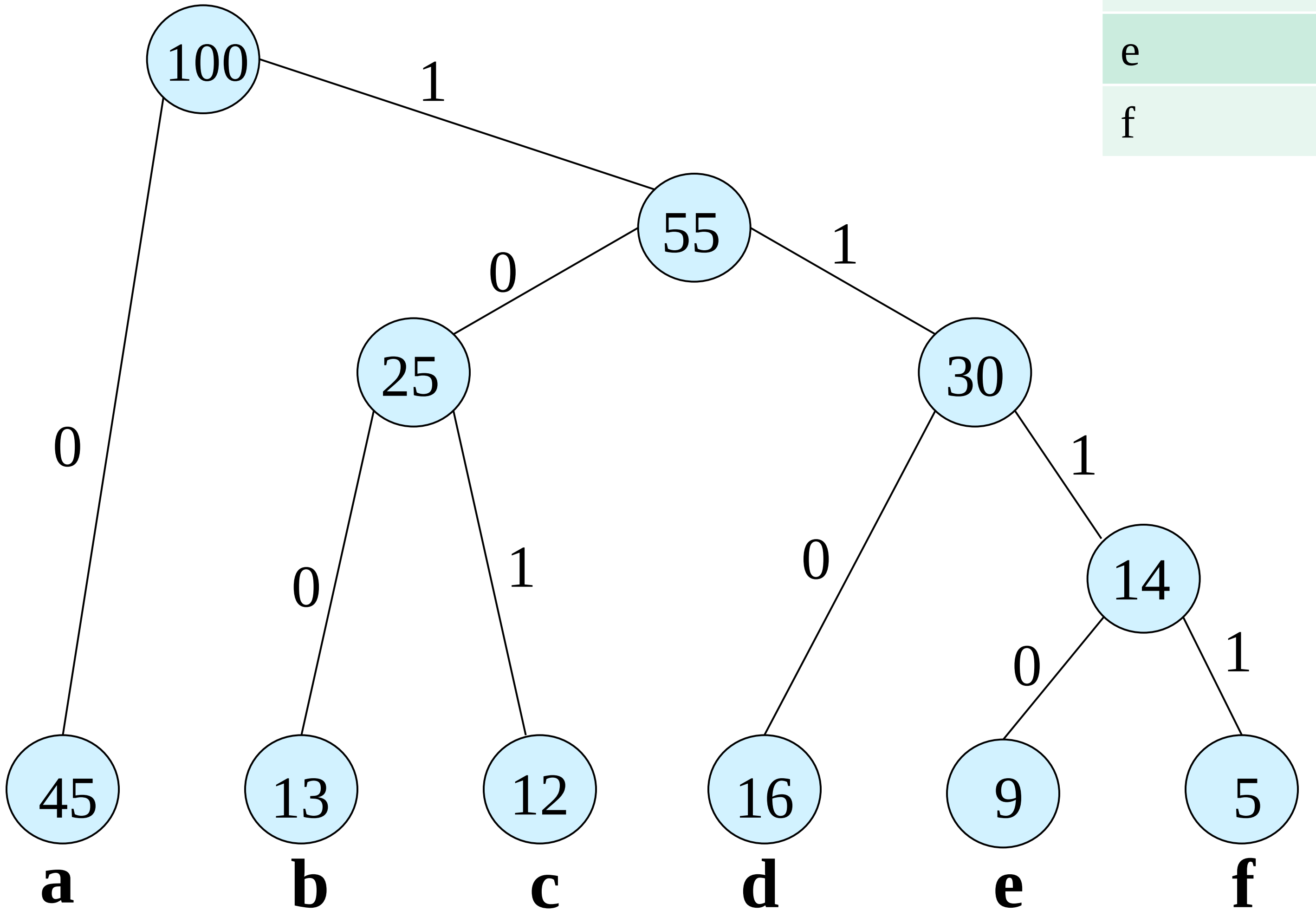
Exemple :

caractère	fréquence
a	45
b	13
c	12
d	16
e	9
f	5



Exemple :

caractère	fréquence
a	45
b	13
c	12
d	16
e	9
f	5



Quiz

Quelle est la complexité de l'algorithme de construction de l'arbre ?

Pour chaque caractère a_i créer une feuille de poids $f(a_i)$
Tant qu'il existe plusieurs arbres :
    On choisi deux arbres T_1 et T_2 de poids minimal.
    On créé un arbre de racine poids(T_1) + poids (T_2) et ayant comme sous-arbre T_1 et T_2 .

On suppose qu'il y a x caractères dans le texte.

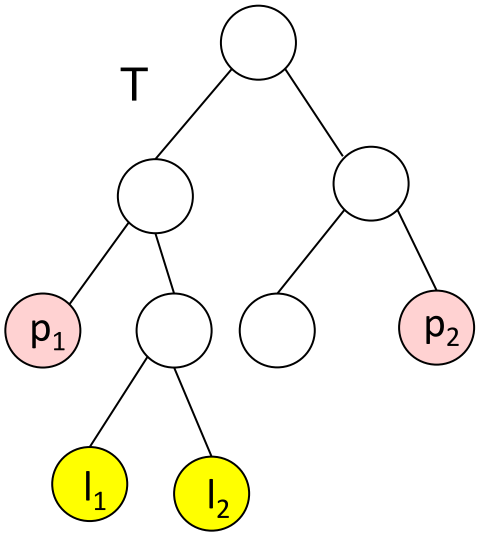
A. $O(x^2)$
B. $O(x \log x)$
C. $O(x)$

Optimalité du codage de Huffman : principe de choix glouton

Soit p_1 et p_2 les caractères de plus faibles fréquences. Montrons que :

il existe toujours un arbre de codage optimal où p_1 et p_2 sont enfants d'un même nœud.

Soit T un arbre de codage optimal, avec l_1 et l_2 enfants d'un même nœud de profondeur max. On suppose que $f(l_1) \leq f(l_2)$ et $f(p_1) \leq f(p_2)$.



Soit prof(i) la profondeur de i dans T (= longueur du code associé à i).

Soit T' l'arbre obtenu en échangeant p_1 et l_1.
Puisque $f(p_1) \leq f(l_1)$ et $\text{prof}(p_1) \leq \text{prof}(l_1)$, le code issu de T' ne sera pas plus long que le code issu de T :

$$\text{Longueur}(T') = \text{Longueur}(T) - f(l_1)\text{prof}(l_1) - f(p_1)\text{prof}(p_1) + f(l_1)\text{prof}(p_1) + f(p_1)\text{prof}(l_1)$$

$$= \text{Longueur}(T) - (f(l_1) - f(p_1))(\text{prof}(l_1) - \text{prof}(p_1)) \leq \text{Longueur}(T)$$

T' est donc optimal. De même, en échangeant l_2 et p_2 dans T', on obtient un arbre de codage T'' optimal.

Propriété de sous-structure optimale

Trouver un arbre optimal pour le problème P

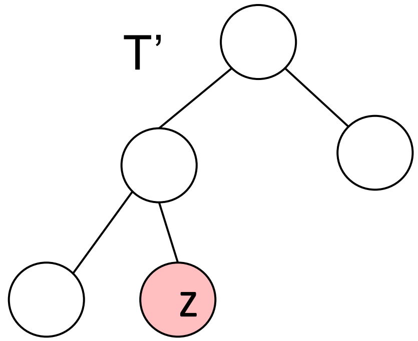
a : 12 b : 4 c : 1 d : 2 e : 2 f : 4

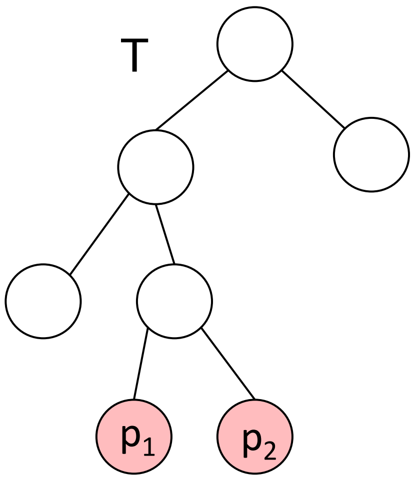
revient à trouver un arbre optimal pour le problème P' (obtenu après la fusion des arbres de plus petits poids)

a : 12 b : 4 z1 : 3 e : 2 f : 4

Il suffit de montrer qu'à tout arbre optimal pour P’ correspond un arbre optimal pour P.

- Soit p_1 et p_2 les caractères de plus petite fréquence dans P
- Soit T’ un arbre de codage optimal pour P’ (où un caractère z de fréquence $f(z) = f(p_1) + f(p_2)$ remplace p_1 et p_2
- Soit T l'arborescence dérivée de T’ en remplaçant la feuille z par un nœud parent de p_1 et p_2 .



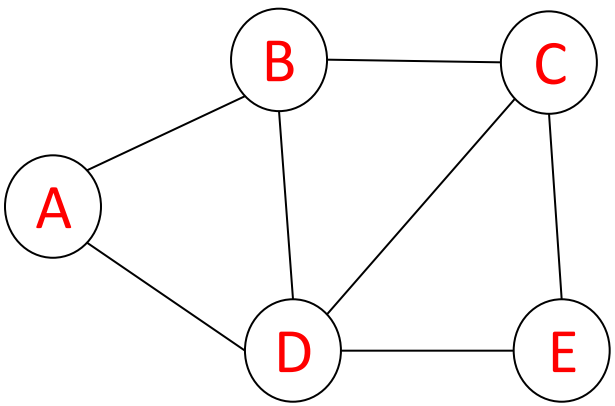


T' optimal pour P’ ⇒ T optimal pour P

Preuve : par l'absurde, supposons que T n'est pas optimal :
∃ U pour P t.q. Longueur(U) < Longueur(T).
Soit U’ l'arbre pour P’ obtenu à partir de U en remplaçant le nœud interne avec les deux enfants p_1 et p_2 par une feuille z.
On a Longueur(U) = Longueur(U') + $f(p_1) + f(p_2)$ et
Longueur(T) = Longueur(T') + $f(p_1) + f(p_2)$.
Ainsi : Longueur(U) < Longueur(T) ⇒ Longueur(U’) < Longueur(T’).
Contradiction avec l'optimalité de T’.

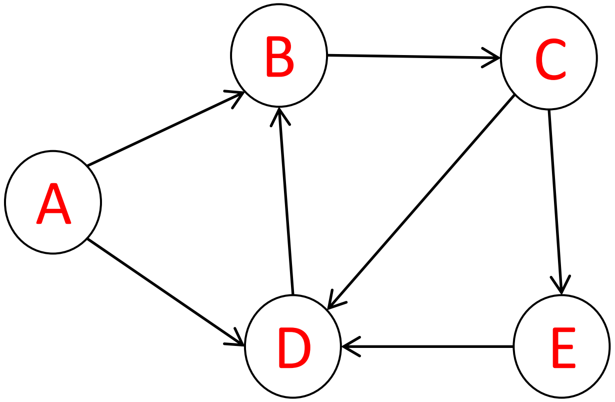
Révisions : Quiz

Dans le graphe suivant, {A,B} est :

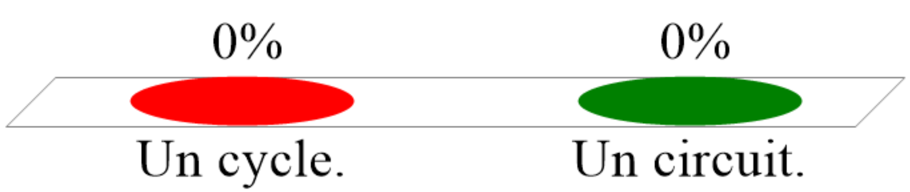


A. Un arc.
B. Une arête.
C. Un chemin.

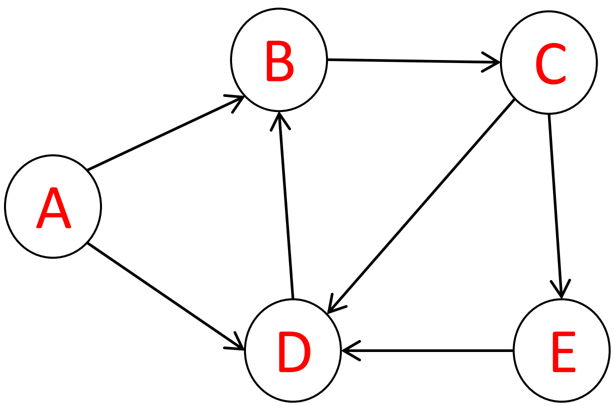
Dans le graphe suivant, (B,C,D,B) est :



A. Un cycle.
B. Un circuit.



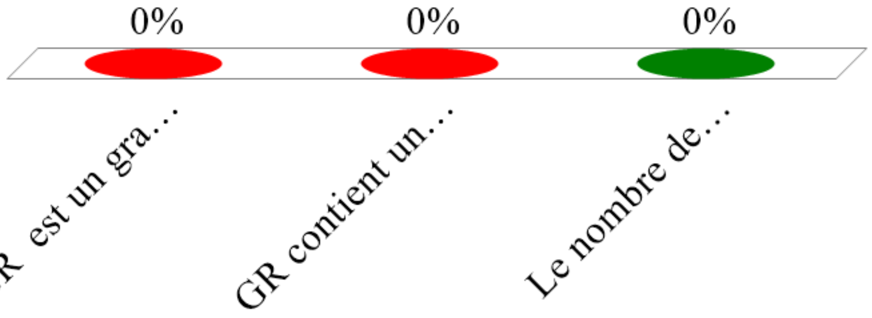
Combien de sommets y a-t-il dans le graphe réduit du graphe suivant?



A. 1
B. 2
C. 3
D. 5

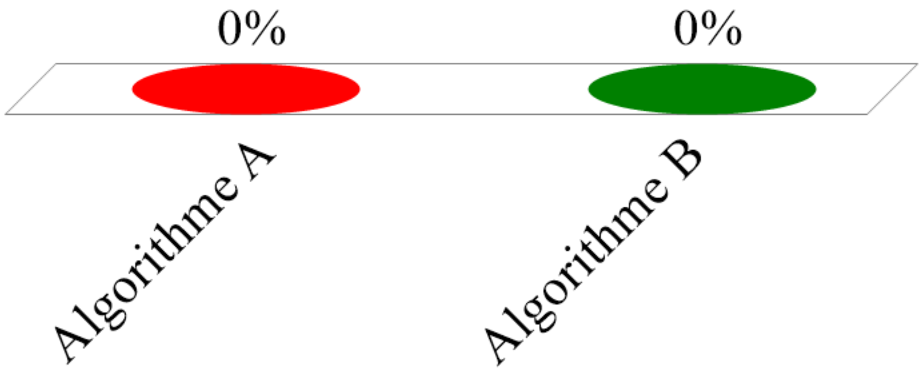
Soit G un graphe fortement connexe et soit G_R le graphe réduit de G. Quelle proposition est fausse ?

A. G_R est un graphe sans circuit.
B. G_R contient un seul sommet.
C. Le nombre de sommets de G_R dépend de G.



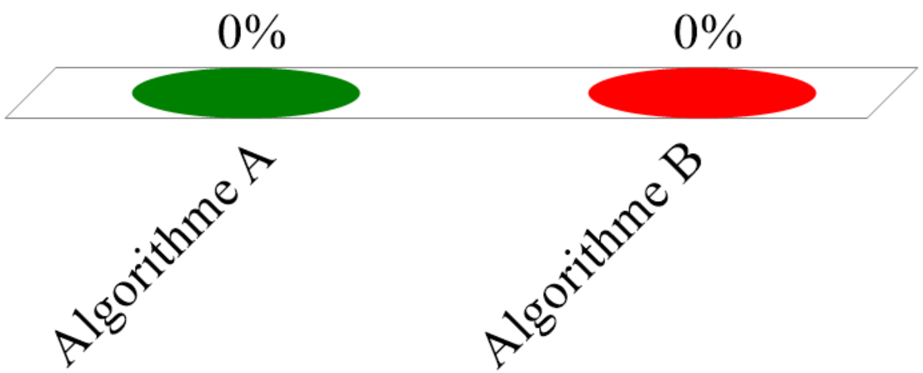
Soient A et B deux algorithmes résolvant le même problème pour un graphe G non orienté.
A est en O(n+m) et B est en O(n log n).
Quel algorithme utiliser si G est un graphe complet ?

A. Algorithme A
B. Algorithme B



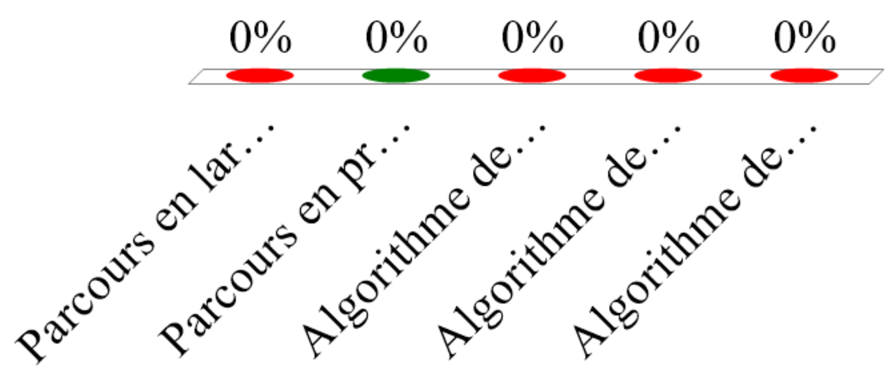
Soient A et B deux algorithmes résolvant le même problème pour un graphe G non orienté.
A est en $O(n+m)$ et B est en $O(n \log n)$.
Quel algorithme utiliser si G est un arbre ?

A. Algorithme A
B. Algorithme B



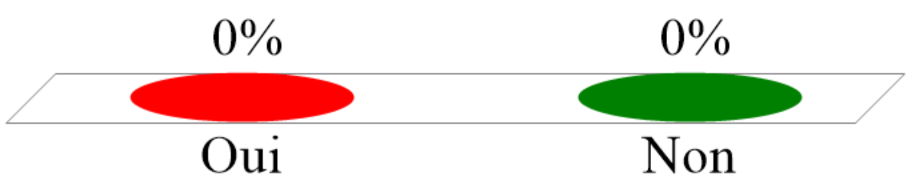
Quel algorithme utiliser pour détecter un circuit dans un graphe orienté ?

A. Parcours en largeur
B. Parcours en profondeur
C. Algorithme de Kruskal
D. Algorithme de Dijkstra
E. Algorithme de Bellman Ford



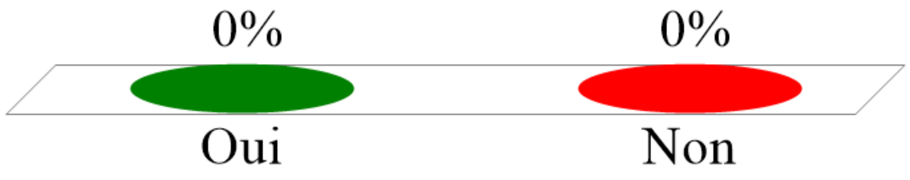
Soit G un graphe connexe, non orienté et valué. Soient u et v deux sommets. Existe-t-il nécessairement une plus courte chaîne entre u et v ?

A. Oui
B. Non



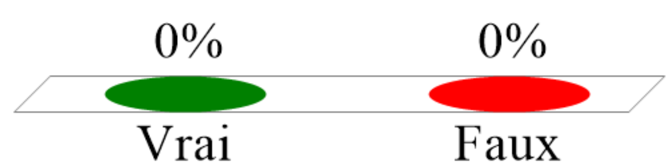
Soit G un graphe connexe, non orienté et valué. Existe-t-il nécessairement un arbre couvrant de coût minimum ?

A. Oui
B. Non



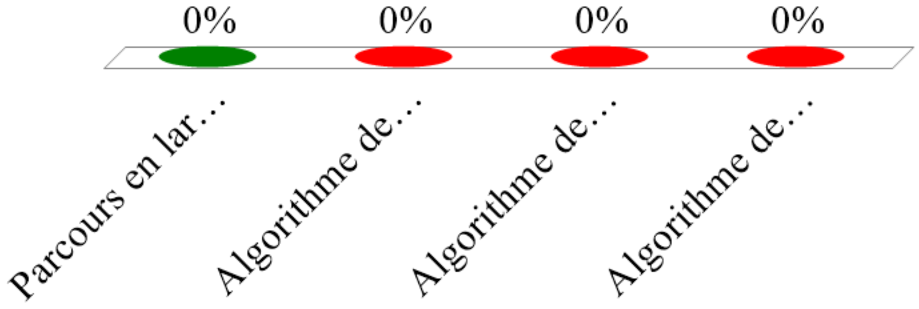
Supposons que l'on ait calculé l'arborescence des plus courts chemins de racine s d'un graphe G valué. Si on modifie le graphe tel que le coût de chaque arête est le double de son coût original. L'arborescence des plus courts chemins reste inchangée.

A. Vrai
B. Faux



Quel algorithme utiliser pour trouver une plus courte chaîne dans un graphe où toutes les arêtes ont le même coût (positif) ?

A. Parcours en largeur
B. Algorithme de Dijkstra
C. Algorithme de Bellman
D. Algorithme de Bellman Ford

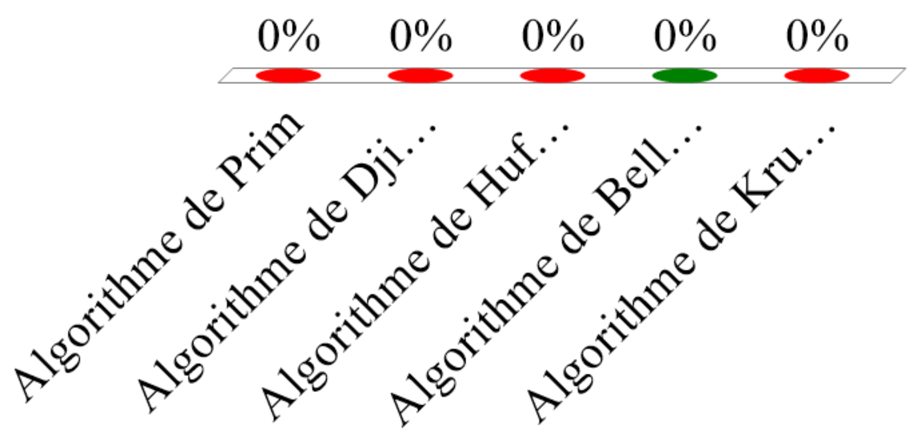


Quel algorithme utiliser pour trouver une plus courte chaîne dans un graphe sans circuit où les coûts des arêtes sont positifs ?

A. Parcours en largeur
B. Algorithme de Dijkstra
C. Algorithme de Bellman
D. Algorithme de Bellman Ford

Parmi les algorithmes suivants, lequel n’est pas un algorithme glouton ?

A. Algorithme de Prim
B. Algorithme de Djikstra
C. Algorithme de Huffman
D. Algorithme de Bellman Ford
E. Algorithme de Kruskal



Quiz

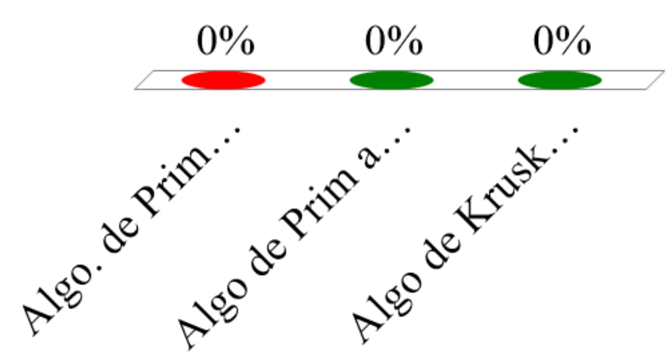
Soit G un graphe connexe. Quel algorithme choisir pour trouver un arbre couvrant de coût minimum de G si G est peu dense (m = 10 n) ?

A. Algo. de Prim sans tas

B. Algo de Prim avec tas

C. Algo de Kruskal avec union pondérée et compression de chemins

- Prim sans tas : $O(n^2)$
- Prim avec tas : $O(m \log n)$
- Kruskal :
  - cas général : $O(m \log m)$
  - coûts des arêtes en $O(n+m)$: $O(n + m\,\alpha(n,m))$.



Quiz

Quel algorithme choisir pour trouver un arbre couvrant de coût minimum de G si G est très dense ($m = 0.2\ n^2$) ?

A. Algo. de Prim sans tas

B. Algo de Prim avec tas

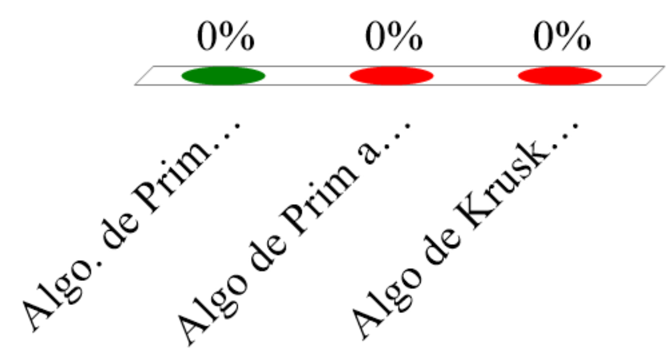
C. Algo de Kruskal avec union pondérée et compression de chemins

- Prim sans tas : $O(n^2)$

- Prim avec tas : $O(m \log n)$

- Kruskal :

- cas général : $O(m \log m)$
- coûts des arêtes en $O(n+m)$: $O(n + m\,\alpha(n,m))$.



Quiz

Quel algorithme choisir pour trouver un arbre couvrant de coût minimum de G si G est peu dense et les coûts des arêtes sont inférieurs à 10m ?

A. Algo. de Prim sans tas

B. Algo de Prim avec tas

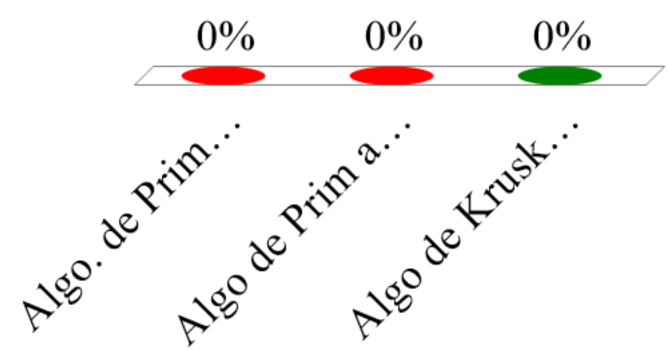
C. Algo de Kruskal avec union pondérée et compression de chemins

- Prim sans tas : $O(n^2)$

- Prim avec tas : $O(m \log n)$

- Kruskal :

- cas général : $O(m \log m)$
- coûts des arêtes en $O(n+m)$: $O(n + m\,\alpha(n,m))$.



Un groupe de TD comporte n personnes (n est pair). Pour le projet, chacun doit se mettre en binôme avec un autre étudiant. Combien y-a-t-il de façons (f(n)) de former des groupes différents ?

(exemple: si n=4, il faut retourner 3).

A. $f(n) = 2\ f(n-1)$
B. $f(n) = (n - 1)\ f(n - 2)$
C. $f(n) = f(n - 1) + (n - 1)\ f(n - 2)$
D. $f(n) = 2^n$