Le problème de Fusion-Recherche

- Etant donné un nombre n d'objets (disons les entiers $1 \dots n$)

- Au départ chaque objet constitue un ensemble

- Deux opérations: FUSIONner deux ensembles, CHERCHer dans quel ensemble se trouve un objet donné
(on peut supposer que nous sommes libres à choisir comme nom d'un ensemble produit le nom qui nous convient parmi les deux noms des ensembles fusionnés)

- Quel est le temps de faire une suite d'opérations?

Une Méthode naive: temps linéaire

- Quand on fait la fusion de A et B, choisir comme nom celui de A et parcourir B en changeant le nom associé à chaque élément de B

- nécessite que les éléments de chaque ensemble sont reliés (par exemple dans une liste avec pointeurs)

- dans le pire cas on fusionne toujours un A d'un seul élément avec un grand B

- donc temps $n - 1$ pour la dernière fusion

- La moyenne? dépend de l'application

Une astuce simple: temps (amorti) $O(\log n)$

- Toujours choisir comme nom celui de l'ensemble le plus grand
- nécessite le stockage de la taille de chaque ensemble
- facile à mettre à jour la taille après une fusion
- un objet est renommé au plus $\log i$ fois dans une suite de i fusions
- temps (amorti) par opération $O(\log n)$

Une astuce de plus: temps (pire cas) $O(\log n)$

- Nouvelle structure de données: un arbre où les pointeurs sont orientés vers la racine: la racine contient le nom de l'ensemble

- toujours fusionner selon les deux tailles: le plus grand des deux ensembles devient père de l'autre: ne pas chercher à mettre à jour les noms (en fait ne plus les stocker)

- pour trouver le nom d'un objet, suivre les pointeurs à la racine

- en montant vers la racine pour trouver le nom, la taille des ensembles double (au moins) à chaque itération

- donc au plus $\log n$ fois

Et encore une

- Chaque fois qu'on a parcouru un chemin d'un objet vers la racine, on le parcourt une deuxième fois, remplacant chaque pointeur par un raccourci directement à la racine !

- Le pire cas d'une opération reste $O(\log n)$

- Mais c'est clair que chaque recherche rend plus vite des recherches à venir

- Et l'effet sur le temps total ?

Les fonctions $\log^*$ et T

- $\log^*(n)$ une fonction qui croît TRES lentement avec n

- le nombre de fois qu'il faut appliquer $\log_2$ à n (avec arrondi vers le bas chaque fois) pour arriver à 1

- $T(n)$ (mon nom privé pour ce cours) une fonction qui croît TRES vite avec n

- la valeur d'une tour de n fois 2

- $\log^*$ est l'inverse de T

- on va démontrer que le temps (amorti) d'une opération de fusion/recherche est $O(\log^*(n))$

Majorer le temps amorti: (1) les gros sauts de taille

- Le temps d'une fusion est constante : on ne fait qu'ajouter un pointeur et sommer les deux tailles

- le temps de suivre un pointeur en cherchant la racine est "facturé" à l'opération de recherche ou à l'opération de fusion qui a ajouté l'objet comme un fils;
selon les tailles des ensembles père et fils

- si les deux tailles ont des valeurs différentes de $\log^*$, facturer à la recherche

- le nombre facturés à une recherche est forcément $O(\log^*(n))$

Majorer le temps amorti: (2) les petits sauts

- si les deux tailles ont la même valeur de $\log^*$, facturer à la fusion

- pour un ensemble de taille s, où $T(i) <= s <= T(i+1)$, le nombre facturé est au plus $\log_2 T(i+1)$, c'est-à-dire $T(i)$ et donc $<= s$

- deux sous-arbres sont disjoints si le rapport entre leurs deux tailles (grand:petit) est inférieur à 2

- donc, pour les s tels que $T(i) <= s < 2T(i)$, total facturé $<= n$

- et pour ceux tels que $2T(i) <= s < 4T(i)$, $<= n/2$

- etc. et pour les s tels que $T(i) <= s < T(i+1)$, $<= 2n$

- et enfin, pour tous les s, $<= 2n \log^*(n)$

Conclusion

- temps total
  $O((nombre\ de\ fusions + nombre\ de\ recherches) *$
  $\log^*(n))$

- temps amorti de chaque opération $O(\log^*(n))$

- pas évident que ce majorant soit strict (pour cet algorithme ou un autre)