EA

Introduc°

Un algorithme est une suite finie d'instruc° q prend des données en entrée et donne en sortie un résultat, en un tps fini.
↳ un algo est correcte, s'il donne la bonne solu° pour chaq instance du pb

Un programme est une suite d'instruc° écrit ds un langage. Il décrit un algo

Efficacité d'un algo

- Complexité en tps : nb d'op. elem^ts
  ↳ (assigna°, comparaisons, accès à un elem^t tab)
  ↳ part aléatoire : traduc°, environm^t (memoire, syst^m, opti...)
- ——— spatiale : l'espace mémoire requis par calcul

↳ pratiq : mesure précise pour un modèle donné de machine
↳ theoriq : ordre de grandeurs de ces coûts

- un algo est efficace si sa complexité est au plus un polynome en n (taille données en entrée)

Les tris

- Par selec° :

```
n ← t.length
pour i ← 0 à n-2 faire
    min ← i
    pour j ← i+1 à n-1 faire
        si T[min]>T[j] alors
            min ← j
    echanger T[i] T[min]
```

=> trouver le min et le placer au debut de la liste triée : $O(n^2)$

- un algo de tri est stable, s'il préserve l'ordonnancem^t initial des élém^ts égaux
- ——— en place, s'il effectue directem^t dans la structure initiale de données & l'espace sup. ne dépend pas de la taille du tab

- Par insert° :

- fonc° triPartiel (T, i)

```
j ← i
tant q j > 0 & T[j] < T[j-1] faire
    échanger T[j-1] & T[j]
    j ← j-1
```

- fonc° triPar insert° (T)

```
n ← t.length
pour i ← 1 à n-1 faire
    triPartiel (T, i)
```

=> On garde le debut du tab trié & y insère les elem^ts q arrive

- dans le pire des cas : $O(n^2)$
- dans le meilleur des cas : $O(n)$

Fonc° recursive

- factorielle (n)

```
si n = 0 alors : retourner 1
sinon retourner n * factorielle(n-1)
```

- logarithme (n)

```
si n ≤ 1 alors : retourner 0
sinon : retourner 1 + logarithme (n/2)
```

une fonc° recursive terminale : la dernière instruc° est un appel récursif

Dichotomie

- fonc° dich (t, x, g, d)

```
if (g <= d)
    m = (g+d)/2
    if (t[m] == m) return m
    if (t[m] > x) return dich(t, x, g, m-1)
    else return dich (t, x, m+1, d)
else return -1
```

Complexité : $O(\log n)$
- chaq tour : $\frac{n}{2}$
=> $\frac{n}{2^k}$
=> $1 \leq \frac{n}{2^k}$ <=> $2^k \leq n$
<=> $k \leq \log_2 n$

listes chainée

- taille flexible ms accès difficile au i^{eme} élem^t

```
insert_head (L, x)
    c = new cell ()      c(key, next)
    c.key = x
    c.next = L.head
    L.head = c
```
(L : liste, c : obj ; L.head : pointeur vers la 1ère cellule)

```
remove_head (L)
    c = L.head
    if c != null
        L.head = c.next
    return c
```

doublem^t ch^n

```
supprimer (L, x)
    si x.prev ≠ null
        x.prev.next ← x.next
    sinon
        L.head ← x.next
    si x.next ≠ null
        x.next.prev ← x.prev
```

```
insert (L, x, y)
    y.next ← x.next
    y.prev ← x
    si x.next ≠ null
        x.next.prev ← y
    x.next ← y
```

files

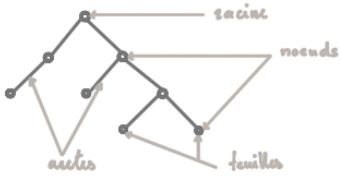
- premier entré, premier sorti

```
empty (L)
    return L.head == null

get (L)
    c (L.head.x, null)
    L.head = L.head.next
    L.head.prev = null
    return c
```

```
put (L, x)
    c (x, null)
    si L.head == null
        L.head = c
    sinon put (L.head, c)

put (c, c1)
    si c.next == null
        c.next = c1
    sinon put (c.next, c1)
```

EA

Arbre binaire



parcours

- prefixe : on part de la racine et affiche tout ce qu'on croise [prio g > d]
- postfixe : on part de la racine et on affiche qnd il n'y a pas de feuille (ou q' ont déjà été affiché) [prio g > d]

Arbre binaire de recherche (ABR)

- val de gauche ≤ val racine
- val de droite > val racine

```
estDansABR(t, e)                                   (e : élém°)
    si estVide(t) => faux
    sinon si e = Val(t) => vrai
    sinon si e ≤ Val(t) => eDA(SAG(t), e)
    sinon => eDA(SAD(t), e)
```

complexité : hauteur(t)

```
insert°(t, e)
    si estVide(t) => Noeud(e, Vide(), Vide())
    sinon si e ≤ Val(t) => Noeud(Val(t), ins(SAG(t), e), SAD(t))
    sinon => Noeud(Val(t), SAG(t), ins(SAD(t), e))
```

Arbre generaux

- Noeud(5, []) = (5)
- Noeud(6, Noeud(5, [])) = (6)(5)
- Noeud(5, [(1)(7)(6), (1)(3)(2)]) = (5)(1)(1)(7)(3)(2)(6)
- Val((1)(3)(2)) = 1
- Foret((5)(1)(1)(7)(3)(2)(6)) = [(1)(7)(6), (1)(3)(2)]

code de Prüfer

```
res ← []
tq (Foret(t) ≠ [])
    x ← val feuille min
    y ← val pere de x
    res ← res.y
    t ← t.sup(x)
renvoyer res
```

AG → AB

- prefixe(t) = prefixe(t(t))
- suffixe(t) = infixe(t(t))

Arbre bin^r parfait (ABP)

- arbre bin^r
- toutes les rangées sont pl^ne sauf la dern^re
- G → D

Tas bin^r max

- ABP
- val noeud > val de ses enfants
- op : SupMax, Ajout => O(log n)

on affiche la racine puis la remplace par la dern^re feuille, puis on réarrange l'arbre

```
triTas(T)
    t ← tasVide()
    pour i ← 0 à n-1 faire
        tasAjout(t, T[i])
    pour i ← n-1 à 0 faire
        T[i] = tasSupMax(t)
    retourner t
```