Recherche séquentielle

Idée. Balayer le tableau jusqu'à trouver x.

```
bool trouve=false; int i=0;
while (i<n && !trouve) {
  if (T[i]==x) trouve=true; else i++;
}
return trouve;
```

Complexité : pire/moyenne $\mathcal{O}(n)$, meilleur $\mathcal{O}(1)$ si x en tête.

Recherche dichotomique (binaire)

Précondition : tableau trié.

```
int g=0, d=n-1;
while (g<=d){
  int m=(g+d)/2;
  if (T[m]==x) return true;
  if (T[m]<x) g=m+1; else d=m-1;
}
return false;
```

Complexité : $\mathcal{O}(\log n)$.

Tableau vs. Liste chaînée

Tableau	Accès aléatoire $\mathcal{O}(1)$, taille fixe
Liste chaînée	Insertion/suppression en tête $\mathcal{O}(1)$, accès séquentiel

Piles et files

Pile (LIFO)

- *Tableau*: push/pop en fin $\mathcal{O}(1)$ amorti
- *Liste*: push/pop en tête $\mathcal{O}(1)$

File (FIFO)

- *Tableau*: deux indices (tête/fin), opérations $\mathcal{O}(1)$
- *Liste*: pointeur de fin, enq/deq $\mathcal{O}(1)$

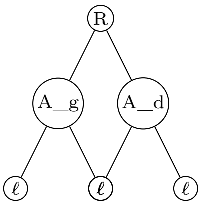
Arbres

Déf. Nœud racine et sous-arbres (éventuellement vides).

Parcours : préordre (racine, A_g, A_d), infixe (A_g, racine, A_d), postordre (A_g, A_d, racine).

Hauteur h (nœuds n) : $\lfloor \log_2 n \rfloor \le h \le n-1$.

Schéma AB (ex.)



Tri : borne inférieure

Tout tri par comparaisons $\Rightarrow \Omega(n \log n)$ au pire.

Tri par sélection

Principe : mettre le plus petit au début à chaque passe.

```
for (i=1..n-1){
  k=i;
  for (j=i+1..n) if (T[j]<T[k]) k=j;
  swap(T[i],T[k]);
}
```

Complexité : $\mathcal{O}(n^2)$ (pire/moyen/meilleur).

Tri par insertion

Insérer $T[i]$ dans le préfixe trié.

```
for (i=2..n){
  x=T[i]; j=i-1;
  while (j>=1 && T[j]>x){ T[j+1]=T[j];
     j--; }
  T[j+1]=x;
}
```

Complexité : pire $\mathcal{O}(n^2)$, meilleur $\mathcal{O}(n)$.

Tri rapide (Quicksort)

Partitionner & trier récursivement.

```
q = partition(T,l,r);
quicksort(T,l,q-1); quicksort(T,q+1,r);
```

Complexité : moyenne $\mathcal{O}(n \log n)$, pire $\mathcal{O}(n^2)$.

Tri par ABR

Insérer dans un ABR puis parcours infixe $\Rightarrow$ séquence triée.

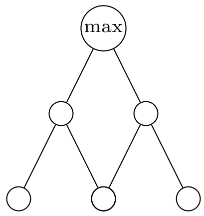
Construction moyenne $\mathcal{O}(n \log n)$ (pire $\mathcal{O}(n^2)$), parcours $\mathcal{O}(n)$.

Tas (Heap)

Représentation tableau

Indices $1..n$: gauche$(i) = 2i$, droite$(i) = 2i+1$, parent$(i) = \lfloor i/2 \rfloor$.

Schéma de tas



Opérations : *heapify* $\mathcal{O}(\log n)$, build-heap $\mathcal{O}(n)$, insertion/extract-max $\mathcal{O}(\log n)$.

Tri par tas : $\mathcal{O}(n \log n)$.

Hachage

But : appartenance de mots dans une table m cases.

Fonction $h : U \to \{0, \ldots, m-1\}$, facteur de charge $\alpha = n/m$.

Chaînage : coût attendu $\mathcal{O}(1+\alpha)$ pour recherche/insertion; pire $\mathcal{O}(n)$.

Principe de recherche

1. Calculer $i = h(x)$
2. Parcourir la liste du seau i
3. Trouvé $\Rightarrow$ succès sinon échec

Gestion des collisions

- Listes chaînées par seau
- Adressage ouvert (linéaire, quadratique, double hachage)

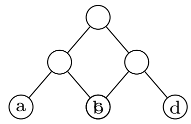
Algorithme de Huffman

Objectif : code préfixe minimisant $L = \sum_c \operatorname{occ}(c)\, \ell(c)$.

Construction : file de priorité fusionnant toujours les 2 plus petites fréquences.

Complexité : $\mathcal{O}(k \log k)$ pour k caractères.

Schéma (exemple)



Propriété de préfixe : aucun code n'est préfixe d'un autre.

Récapitulatif des complexités

Recherche séquentielle	$\mathcal{O}(n)$
Recherche binaire	$\mathcal{O}(\log n)$
Tri sélection	$\mathcal{O}(n^2)$
Tri insertion	pire $\mathcal{O}(n^2)$, meilleur $\mathcal{O}(n)$
Quicksort	moy. $\mathcal{O}(n \log n)$, pire $\mathcal{O}(n^2)$
Tri par tas	$\mathcal{O}(n \log n)$
Tri via ABR	moy. $\mathcal{O}(n \log n)$, pire $\mathcal{O}(n^2)$
Tas (build / op.)	$\mathcal{O}(n)$ / $\mathcal{O}(\log n)$
Hachage (attendu)	$\mathcal{O}(1)$ par op. si α borné
Huffman	$\mathcal{O}(k \log k)$

Graphes (orientation, bases)

Graphe fini simple orienté : $G = (X, A)$, X ensemble fini de sommets, $A \subseteq X \times X$ sans boucle ni multi-arcs.
Degrés : $\deg^+(x) = |\{(x, y) \in A\}|$, $\deg^-(x) = |\{(y, x) \in A\}|$.

$$n = |X|, \qquad m = |A|, \qquad \sum_{x\in X} \deg^+(x) = \sum_{x\in X} \deg^-(x) = m.$$

Chaîne (chemin) : $(x_0, \ldots, x_k)$ avec $(x_i, x_{i+1}) \in A$.
Simple : pas de sommet répété; **circuit** : $x_0 = x_k$; **élémentaire** : simple et sans répétition d'arcs.
Fortement connexe : $\forall x, y$ chaîne de x vers y.
Composante fortement connexe (CFC) : classe d'équivalence par la relation "$x \rightsquigarrow y$ et $y \rightsquigarrow x$".
Partition en CFC
Soient $C_1, \ldots, C_k$ les CFC de G :

1. $X = \bigsqcup_{i=1}^{k} C_i$
2. Si $i \neq j$, $C_i \cap C_j = \varnothing$
3. Le graphe des CFC (condensation) est un DAG.

Représentations
Matrice d'adjacence $M = (m_{xy})$ avec $m_{xy} = 1$ si $(x, y) \in A$, sinon 0.
Listes d'adjacence : pour chaque x, la liste de ses successeurs (ou prédécesseurs).
Coût mémoire : matrice $\Theta(n^2)$, listes $\Theta(n + m)$.

Plus court & plus long chemin

Plus court chemin (pcc). Longueur = somme des poids $w(a)$.
Absorbant (négatif) : un circuit de poids < 0 $\Rightarrow$ pas de pcc.
Plus long chemin. NP-difficile en général; dans un DAG on peut le faire en ordre topologique (changer min $\to$ max).
Dijkstra ($w \geq 0$)
Hyp. poids non négatifs. **Idée :** invariant d'ensemble S des sommets "figés".

```
Init:  d[s]=0; d[x]=+inf pour x!=s; S={}
Tant que S!=X:
  u = argmin_{x in X\S} d[x]
  S = S U {u}
  pour (u,v) in A:
    si d[v] > d[u] + w(u,v):
       d[v] = d[u] + w(u,v); pere[v]=u
```

Complexité : $\mathcal{O}(n^2)$ avec matrice; $\mathcal{O}((n + m)\log n)$ avec tas binaire.
Bellman-Ford (poids quelconques)
Détecte circuits négatifs; calcule les pcc si aucun accessible à s.

```
Init: d[s]=0, d[x]=+inf sinon
Repeter n-1 fois:
  pour chaque arc (u,v):
    d[v] = min(d[v], d[u] + w(u,v))
Verif circuit negatif:
  s'il existe (u,v) avec d[v] > d[u] +
     w(u,v) -> negatif
```

Complexité : $\mathcal{O}(n\,m)$.

Graphe sans circuit (DAG)

Numérotation/ordre topologique
Déf. Bijection $\tau : X \to \{1, \ldots, n\}$ telle que $(x, y) \in A \Rightarrow \tau(x) < \tau(y)$.
Propriété : G est un DAG $\Leftrightarrow$ un ordre topologique existe.
Calcul d'un ordre topo (Kahn)

```
Entree: DAG G
S = {sommmets d'indegre 0}; ordre=[]
Tant que S non vide:
  u = extraire(S)
  ordre.ajouter(u)
  pour (u,v) in A:
    enlever (u,v); si indegre(v)==0:
     S.ajouter(v)
```

Complexité : $\mathcal{O}(n + m)$.
Pcc dans un DAG (Bellman "topo")

```
Entree: DAG, ordre topologique t(1..n)
Init d[s]=0; d[x]=+inf sinon; pere[x]=nil
Pour i=1..n (dans l'ordre topo):
  u = t[i]
  pour (u,v) in A:
    si d[v] > d[u] + w(u,v):
       d[v] = d[u] + w(u,v); pere[v]=u
```

Complexité : $\mathcal{O}(n + m)$.

Graphe non orienté

Arêtes non ordonnées $\{x, y\}$. $G = (X, E)$, $n = |X|$, $m = |E|$.
Degré $d(x)$ nombre d'arêtes incidentes à x.

$$\sum_{x\in X} d(x) = 2m \quad \text{(handshaking).}$$

Chaîne : $(x_0, \ldots, x_k)$ avec $\{x_i, x_{i+1}\} \in E$.
Cycle : $x_0 = x_k$, $k \geq 3$.
Connexité : $\forall x, y$ chaîne entre x et y. Les composantes connexes $C_1, \ldots, C_r$ partitionnent X.
Forêt/Arbre couvrant : sous-graphe (X, F), acyclique, connexe, $|F| = n - 1$.
Pont (arête isthme) : arête dont la suppression augmente le nombre de composantes.
Conséquences

- Si G est connexe, alors $m \geq n - 1$.
- G connexe $\Rightarrow$ il existe un arbre couvrant.
- Un cycle $\Rightarrow$ on peut retirer une arête du cycle sans perdre la connexité.

Utilisation typique (pères & distances)

Soit un parcours (BFS/DFS) depuis s. On enregistre $d(x)$ (distance/numéro) et $p(x)$ (père).

x	$d(x)$	$p(x)$
s	0	–
...	...	...

Propriété (BFS) : si G non pondéré (ou poids 1), $d(x)$ est la longueur du pcc $s \rightsquigarrow x$ et l'arbre des pères est un arbre des plus courts chemins.

Remarques pratiques

- **Choix de structure :** matrice pour graphes denses; listes pour graphes clairsemés.
- **Chemins longs/courts dans DAG :** même DP sur ordre topo avec max ou min.
- **Détection de cycles :** un ordre topo n'existe pas $\Leftrightarrow$ il y a un cycle.

Lemmes et THM (graphes non orientés)

Lemme 2. Si G est sans cycle alors $m \leq n-1$.
Corollaire. Un arbre a n sommets, $n-1$ arêtes.
Lemme 3. $a \in E$ est un isthme $\Leftrightarrow$ a n'appartient à *aucun* cycle.
THM équivalences pour G connexe :

1. G connexe et $m = n-1$
2. G est sans cycle
3. Il existe une unique chaîne simple entre x et y pour tout $x \neq y$
4. G est connexe et l'ajout d'une arête crée un cycle

Algorithme de Kruskal (ACM poids min)

Principe.

1. Trier les arêtes par poids croissant
2. Parcourir la liste et ajouter l'arête si elle ne crée pas de cycle

Test de cycle : union-find (composantes).
Pseudo-code (schéma).

```
ACM = {}
trier E par poids croissant
pour (u,v) dans E:
  if find(u)!=find(v):
     ACM.add((u,v)); union(u,v)
```

Complexité : $\mathcal{O}(m \log m)$ (tri) $+\mathcal{O}(m\,\alpha(n))$ (UF).
Détection de cycle par CC (version simple)
On maintient $CC[i]$ = numéro de composante du sommet i; fusion si arête choisie.
Complexité illustrée : $\mathcal{O}(m \log m) + n^2$ (naïf).

Algorithme de Prim (ACM)

Principe. Démarre d'un sommet; à chaque étape, ajoute l'arête la moins chère incidente à l'arbre courant.

```
S = {s};  poids[v]=+inf; pere[v]=nil
maj voisins de s
tant que |S|<n:
  u = argmin_{v notin S} poids[v]
  S = S U {u}
  pour (u,w) arete:
    si w notin S et c(u,w) < poids[w]:
       poids[w]=c(u,w); pere[w]=u
```

Complexité : $\mathcal{O}(n^2)$ (matrice), $\mathcal{O}(m \log n)$ (tas binaire).

Parcours d'un graphe

Parcours orienté (schéma "marquer/examiner")

- Au départ, aucun sommet marqué; aucune arête traversée
- Choisir un sommet x; **traverser** (x, y) si non traversée
- Si y non marqué $\Rightarrow$ marquer y, $p\grave{e}re(y) = x$

Boucle sur les arêtes jusqu'à ce que tous les sommets soient examinés.
Coût : $\mathcal{O}(n+m)$.
Parcours non orienté
Même idée avec arêtes $\{x, y\}$; les composantes connexes émergent naturellement.

BFS (largeur) & DFS (profondeur)

Objets. $G = (V, E)$ graphe; $s \in V$ source; Q=file FIFO; $d[v]$=distance (#arêtes) depuis s; $pere[v]$=parent BFS; *marquer* $v \Leftrightarrow visited[v] \leftarrow$ vrai; enqueue(Q, v)/dequeue(Q)=opérations de file; "pour (u, v)" $\equiv$ pour tout arc $(u, v) \in E$.
BFS (file)

```
marquer s; d[s]=0; Q={s}
while Q non vide:
u=dequeue(Q)
pour (u,v):
    si v non marque:
    marquer v; d[v]=d[u]+1;
 pere[v]=u; enqueue(Q,v)
```

Propriété. $d[v]$ = distance minimale (non pondéré). **Complexité.** $\mathcal{O}(n+m)$.
DFS (pile/récursif)

```
DFS(u):
  pre[u]=++time
  pour (u,v):
    si v non visite: pere[v]=u; DFS(v)
  post[u]=++time
```

Numérotations pré/postfixe. $pre[u]$ à l'entrée, $post[u]$ à la sortie. Utile pour CFC/-topo.

CFC (composantes fortement connexes)

Kosaraju (2 DFS).

1. DFS sur G pour obtenir l'ordre décroissant de $post$.
2. DFS sur G^T en suivant cet ordre; chaque arbre découvert $\Rightarrow$ une CFC.

Complexité : $\mathcal{O}(n+m)$.

Flot maximum & coupe minimum

Réseau. $G = (X, A)$ orienté valué par capacités $c : A \to \mathbb{R}_+$; source s, puits t.
Flot. $f : A \to \mathbb{R}$ tel que :

- (Capacité) $0 \leq f(x, y) \leq c(x, y)$
- (Conservation) $\sum_y f(y, x) = \sum_y f(x, y)$ pour $x \neq s, t$

Valeur. $|f| = \sum_y f(s, y) - \sum_y f(y, s)$.
Résiduel. $c_f(x, y) = c(x, y) - f(x, y)$ et arc retour (y, x) de capacité $f(x, y)$.
Théorème (Max-flow / Min-cut)
Pour tout flot f et toute coupe $(S, \bar{S})$ avec $s \in S$, $t \in \bar{S}$:

$$|f| \leq c(S, \bar{S}) = \sum_{x \in S,\, y \in \bar{S}} c(x, y),$$

avec égalité $\Leftrightarrow$ f est maximum et la coupe est minimum.

Ford–Fulkerson (chemins augmentants)

Idée.

1. Initialiser $f = 0$; calculer le résiduel G_f
2. Tant qu'il existe un chemin P de s à t dans G_f:
3. $\Delta = \min\{c_f(e) : e \in P\}$; augmenter f de Δ le long de P

Complexité. Dépend du choix des chemins. Avec BFS (Edmonds–Karp) : $\mathcal{O}(n\,m^2)$.
Arrêt. Aucun chemin augmentant $\Rightarrow$ flot max atteint; les sommets atteignables dans G_f définissent une coupe min.

Notes rapides & rappels

- **Pont/isthme.** Suppression augmente le # de composantes.
- **Arbre couvrant.** $n-1$ arêtes; unique chemin simple entre deux sommets.
- **DAG.** Numérotation topologique $\Leftrightarrow$ pas de cycle.
- **Choix structures.** Matrice $\Theta(n^2)$ (dense), listes $\Theta(n+m)$ (sparse).
- **Coûts typiques.** BFS/DFS $\mathcal{O}(n+m)$; Kruskal $\mathcal{O}(m \log n)$; Prim $\mathcal{O}(m \log n)$; CFC $\mathcal{O}(n+m)$; Edmonds–Karp $\mathcal{O}(nm^2)$.

Coupes, marquages et flot

Coupe $(S,\bar{S})$. $s\in S$, $t\in\bar{S}$. Capacité $c(S,\bar{S}) = \sum_{x\in S,\, y\in\bar{S}} c(x,y)$.
Résiduel. $c_f(x,y) = c(x,y) - f(x,y)$ (arc retour (y,x) de capacité $f(x,y)$).
Algorithme de marquage (après arrêt).

- Marquer s.
- Tant qu'il existe un sommet marqué x et un voisin y non marqué tel que

  $$c_f(x,y) > 0 \text{ (arc direct disponible)} \quad \text{ou} \quad c_f(y,x) > 0 \text{ (arc retour)},$$

  marquer y.

À la fin, $S = \{\text{sommets marqués}\}$: *les arcs traversant de S vers $\bar{S}$ sont saturés et ceux de $\bar{S}$ vers S portent du flot nul.* $(S,\bar{S})$ est une **coupe minimum** et $|f| = c(S,\bar{S})$.
Ford–Fulkerson (rappel succinct)

```
f=0; construire G_f
tant qu'il existe un chemin P de s a t
    dans G_f:
  Delta = min{ c_f(e) : e dans P }
  augmenter f de Delta le long de P
  mettre a jour G_f
```

Complexités usuelles. FF naïf : peut être pseudo-polynomial. Edmonds–Karp (BFS dans G_f) : $\mathcal{O}(nm^2)$.

Application : Couplage maximum biparti

Soit $G = (X\cup Y, E)$ biparti. Construire le réseau :

- arcs $s\to x$ ($x\in X$) de capacité 1 ;
- arcs $x\to y$ pour $(x,y)\in E$ de capacité 1 ;
- arcs $y\to t$ ($y\in Y$) de capacité 1.

Tout flot f correspond à un **couplage** $C = \{(x,y) : f(x,y) = 1\}$. Max-flot = taille du couplage maximum.

Complexité (implémentation simple). $\mathcal{O}(mn)$; avec Hopcroft–Karp : $\mathcal{O}(\sqrt{n}\,m)$.

Théorème de Menger (version arcs)

Pour $a,b\in X$, soient $N_{a,b}$ le # minimal d'arcs à supprimer pour séparer a de b, et $P_{a,b}$ le # maximal de **chemins arc-disjoints** de a à b. Alors

$$N_{a,b} = P_{a,b}.$$

Idée de preuve. Réduction à un réseau : capacité 1 par arc, max-flot = # de chemins arc-disjoints, min-coupe = # d'arcs à retirer. Donc $P_{a,b} = N_{a,b}$.

Taille de codage & problèmes de décision

Une **instance I** est encodée binaire; taille($\mathbf{I}$) = $|\mathbf{I}|$ (en bits).
Un **problème de décision** attend "oui/non". On dit qu'un algorithme est **polynomial** si son temps est majoré par $|\mathbf{I}|^k$ pour un k constant.
Classes P et NP

- **P** : problèmes décidables en temps polynomial.
- **NP** : "vérifiables" en temps polynômial : si la réponse est "oui", il existe un **certificat** y de taille polynomiale vérifiable en $poly(|\mathbf{I}|)$.

On sait $P\subseteq NP$. On ignore si $P = NP$.
Réductions polynomiales
$\Pi_1 \le_P \Pi_2$ s'il existe une transformation polynomiale T telle que

$$\mathbf{I}\in\Pi_1 \iff T(\mathbf{I})\in\Pi_2.$$

Si $\Pi_1 \le_P \Pi_2$ et Π_2 est polynomiale, alors Π_1 l'est aussi.
NP-difficile / NP-complet

- **NP-difficile** : tout problème de NP s'y réduit.
- **NP-complet** : dans NP *et* NP-difficile.

Exemples classiques (NP-complétude)

- **3-SAT** : formule CNF à clauses de taille 3 satisfiable ?
- **PVC** (Vertex Cover) : existe-t-il k sommets couvrant toutes les arêtes ?
- **Stable/IS** : existe-t-il un ensemble stable (indépendant) de taille $\ge k$?
- **Clique** : existe-t-il une clique de taille $\ge k$?

Chaîne de réductions standard :

$$\text{3-SAT} \le_P \text{Clique} \equiv_P \text{Stable} \le_P \text{PVC}.$$

Donc **PVC**, **Stable** et **Clique** sont NP-complets (et de même pour 3-SAT).

Complexité de quelques algorithmes de flots

Ford–Fulkerson (chemins arbitraires)	pseudo-polynomial
Edmonds–Karp (BFS)	$\mathcal{O}(nm^2)$
Dinic (niveaux + blocs)	$\mathcal{O}(n^2m)$
Biparti (Hopcroft–Karp)	$\mathcal{O}(m\sqrt{n})$

Notes rapides de cours

- Dans la coupe min issue du marquage, *les arcs retenus par la forte a-b-connexité* sont ceux franchissant $S\to\bar{S}$.
- Pour numéroter un DAG, on peut numéroter de 1 à n en suivant l'ordre topologique.
- Les tableaux de pères/distances (BFS/DFS) suffisent à reconstruire des chemins.