Algorithmes à connaître

Aubin SIONVILLE

MPI Clemenceau - 2021-2023

Bases

Algorithme de recherche dichotomique

Entrées : un tableau T trié et un élément x
Sortie : l'indice de l'élément x dans le tableau T ou -1 Si x n'est pas dans T
Complexité : $\mathcal{O}(\log n)$

```
Dichotomie (T, x) :
    debut ← 1
    fin ← longueur(T)
    Tant que debut ≤ fin faire
        milieu ← (debut + fin) / 2
        Si T[milieu] = x alors
            Retourner milieu
        Sinon Si T[milieu] < x alors
            debut ← milieu + 1
        Sinon
            fin ← milieu - 1
    Retourner -1
```

Exponentiation rapide

Entrée : un entier n et un entier positif p
Sortie : n^p
Complexité : $\mathcal{O}(\log p)$

```
ExpoRapide (n, p) :
    Si p = 0 alors
        Retourner 1
    Sinon Si p = 1 alors
        Retourner n
    Sinon Si p est pair alors
        Retourner ExpoRapide (n*n, p/2)
    Sinon
        Retourner ExpoRapide (n*n, (p-1)/2) * n
```

Algorithme d'Euclide

Entrées : deux entiers a et b
Sortie : le PGCD de a et b
Complexité : $\mathcal{O}(\log \min(a, b))$

```
PGCD (a, b) :
    Tant que b ≠ 0 faire
        t ← b
        b ← a mod b
        a ← t
    Retourner a
```

Algos de tri

Tri par insertion

Entrée : un tableau T
Sortie : le tableau T trié
Complexité : $\mathcal{O}(n^2)$

```
TriInsertion (T) :
    n ← longueur(T)
    Pour i allant de 2 a n faire
        x ← T[i]
        j ← i - 1
        Tant que j ≥ 1 et T[j] > x faire
            T[j+1] ← T[j]
            j ← j - 1
        T[j+1] ← x
```

Tri par sélection

Entrée : un tableau T
Sortie : le tableau T trié
Complexité : $\mathcal{O}(n^2)$

```
TriSelection (T) :
    n ← longueur(T)
    Pour i allant de 0 a n-2 faire
        mini ← i
        Pour j allant de i+1 a n-1 faire
            Si T[j] < T[mini] alors
                mini ← j
        Si mini ≠ i alors
            Echanger (T[mini], T[i])
```

Tri fusion

Entrée : un tableau T
Sortie : le tableau T trié
Complexité : $\mathcal{O}(n \log n)$

```
Fusion (T1, T2) :
    Si T1 est vide alors
        Retourner T2
    Si T2 est vide alors
        Retourner T1
    Si A[1] ≤ B[1] alors
        Retourner A[1] ⊕ Fusion (A[2..longueur(A)], B)
    Sinon
        Retourner B[1] ⊕ Fusion (A, B[2..longueur(B)])

TriFusion (T) :
    Si longueur(T) ≤ 1 alors
        Retourner T
    Sinon
        m ← longueur(T) / 2
        T1 ← TriFusion (T[1..m])
        T2 ← TriFusion (T[m+1..longueur(T)])
        Retourner Fusion (T1, T2)
```

Tri par tas

Entrée : un tableau T
Sortie : le tableau T trié
Complexité : $\mathcal{O}(n \log n)$

```
Entasser (T, i) :
    l ← Gauche(i)
    r ← Droite(i)
    maxi ← i
    Si l ≤ longueur(T) et T[l] > T[i] alors
        maxi ← l
    Si r ≤ longueur(T) et T[r] > T[i] alors
        maxi ← r
    Si maxi ≠ i alors
        Echanger (T[i], T[maxi])
        Entasser (T, maxi)

ConstruireTas (T) :
    Pour i allant de longueur(T)/2 a 1 faire
        Entasser (T, i)

TriTas (T) :
    ConstruireTas (T)
    Pour i allant de longueur(T) a 2 faire
        Echanger (T[1], T[i])
        longueur(T) ← longueur(T) - 1
        Entasser (T, i)
```

Algos des arbres

Parcours préfixe

Entrée : un arbre A
Sortie : un tableau des sommets de A trié selon un parcours préfixe
Complexité : $\mathcal{O}(|S|)$

```
ParcoursPrefixe (A) :
visites ← []
visites ← visites ∪ {A}
Si filsGauche(A) ≠ ∅ alors
    visites ← visites ∪ ParcoursPrefixe (filsGauche(A))
Si filsDroit(A) ≠ ∅ alors
    visites ← visites ∪ ParcoursPrefixe (filsDroit(A))
Retourner visites
```

Parcours infixe

Entrée : un arbre A
Sortie : un tableau des sommets de A trié selon un parcours infixe
Complexité : $\mathcal{O}(|S|)$

```
ParcoursInfixe (A) :
    visites ← []
    Si filsGauche(A) ≠ ∅ alors
        visites ← visites ∪ ParcoursInfixe (filsGauche(A))
    visites ← visites ∪ {A}
    Si filsDroit(A) ≠ ∅ alors
        visites ← visites ∪ ParcoursInfixe (filsDroit(A))
    Retourner visites
```

Parcours suffixe

Entrée : un arbre A
Sortie : un tableau des sommets de A trié selon un parcours suffixe
Complexité : $\mathcal{O}(|S|)$

```
ParcoursSuffixe (A) :
visites ← []
Si filsGauche(A) ≠ ∅ alors
    visites ← visites ∪ ParcoursSuffixe (filsGauche(A))
Si filsDroit(A) ≠ ∅ alors
    visites ← visites ∪ ParcoursSuffixe (filsDroit(A))
visites ← visites ∪ {A}
Retourner visites
```

Rotation gauche

Entrée : un arbre A
Sortie : un arbre B tel que B est la rotation gauche de A
Complexité : $\mathcal{O}(1)$

```
RotationGauche (A) :
    B ← filsDroit(A)
    filsDroit(A) ← filsGauche(B)
    filsGauche(B) ← A
    Retourner B
```

Rotation droite

Entrée : un arbre A
Sortie : un arbre B tel que B est la rotation droite de A
Complexité : $\mathcal{O}(1)$

```
RotationDroite (A) :
    B ← filsGauche(A)
    filsGauche(A) ← filsDroit(B)
    filsDroit(B) ← A
    Retourner B
```

Algos de graphes

Parcours en profondeur

Entrées : un graphe G=(S,A), un sommet s ∈ S
Sortie : un tableau des sommets de G trié selon un parcours en profondeur
Complexité : $\mathcal{O}(|S| + |A|)$

```
ParcoursProfondeur (G, s) :
    visites ← []
    visites ← visites ∪ {s}
    Pour tout s' ∈ voisins(s) faire
        ParcoursProfondeur (G, s')
```

Parcours en largeur

Entrées : un graphe G=(S,A), un sommet s ∈ S
Sortie : un tableau des sommets de G trié selon un parcours en largeur
Complexité : $\mathcal{O}(|S| + |A|)$

```
ParcoursLargeur (G, s) :
    F ← FileVide
    visites ← []
    visites ← visites ∪ {s}
    Enfiler(F, s)
    Tant que F est non vide faire
        s ← Defiler(F)
        Pour tout s' ∈ voisins(s) faire
            Si s' ∉ visites alors
                visites ← visites ∪ {s'}
                Enfiler(F, s')
```

Tri préfixe

Entrées : un graphe G=(S,A), s ∈ S, Res, Visites Sortie : Modifie Res et Visites pour que Res soit un parcours préfixe de Visites et des sommets accessibles depuis s
Complexité : $\mathcal{O}(|S| + |A|)$

```
ExploreDescendant (G, s, Res, Visites) :
    todo ← PileVide
    Empiler (todo, (s, Succ(s)))
    Visites ← Visites ∪ {s}
    Tant que todo ≠ PileVide faire
        (x,l) ← Depiler(todo)
        Si l = () alors
            Res ← x · Res
        Sinon
            t · l' ← l   #Separe la tete et le reste de la pile
            Empiler (todo, (x, l'))
            Si t ∉ Visites alors
                Visites ← Visites ∪ {t}
                Empiler (todo, (t, Succ(t)))

TriPrefixe (G, s) :
    Visites ← ∅
    Res ← ∅
    Tant que Visites\S ≠ ∅ faire
        s ← un sommet de S\Visites
        ExploreDescendant (G, s, Res, Visites)
    Retourner Res
```

Kosaraju

Entrées : un graphe G=(S,A)
Sortie : les composantes fortement connexes de G
Complexité : $\mathcal{O}(|S| + |A|)$

```
Kosaraju (G) :
    s ← un sommet de S
    T ← TriPrefixe (G, s)
    On parcoure G^T en utilisant l'ordre de T comme points de regeneration.
    On retourne le plus petit partitionnement associe au parcours.
```

Kruskal

Entrées : un graphe G=(S,A) connexe
Sortie : un arbre couvrant de poids minimal de G
Complexité : $\mathcal{O}(|S| + |A|)$

```
Kruskal (G) :
    B ← ∅
    U ← ∅
    Tant que ∃ u,v tq u ≁_B v faire
        {x,y} ∈ A\U de poids minimal
        Si x ∼_B y alors
            U ← U ∪ {{x,y}}
        Sinon
            U ← U ∪ {{x,y}}
            B ← B ∪ {{x,y}}
    Retourner T=(S, B)
```

Couplage biparti

Entrées : un graphe G=(S,A) biparti
Sortie : un couplage maximal de G
Complexité : $\mathcal{O}(|S| + |A|)$

```
    A faire
```

Dijkstra

Entrées : un graphe G=(S,A), un sommet s ∈ S
Sortie : Un tableau des distances minimales de s vers les autres sommets
Complexité : $\mathcal{O}(|A| + |S| \log |S|)$

```
Djikstra (G, s) :
    distance ← [ ∞ pour tout s ∈ S]
    file_priorite ← FilePriorite()
    Enfiler (file_priorite, (0, s))

    Tant que file_priorite ≠ ∅ faire
        (d, u) ← Defiler (file_priorite)

        Pour tout v ∈ Succ(u) faire
            Si distance[v] > distance[u] + Poids(u,v) alors
                distance[v] ← distance[u] + Poids(u,v)
                Enfiler (file_priorite, (distance[v], v))

    Retourner distance
```

Floyd Warshall

Entrées : un graphe G=(S,A)
Sortie : Un tableau des distances minimales entre tous les sommets
Complexité : $\mathcal{O}(|S|^3)$

```
FloydWarshall (G) :
    dist ← 0_{M_{|S|,|S|}}
    Pour tout a = (x,y) ∈ A faire
        dist_{x,y} ← Poids(a)
    Pour tout x ∈ S faire
        dist_{x,x} ← 0
    Pour k allant de 1 a |S| faire
        Pour i allant de 1 a |S| faire
            Pour j allant de 1 a |S| faire
                dist_{i,j} ← min(dist_{i,j}, dist_{i,k} + dist_{k,j})
```

Algos de textes

Boyer-Moore

```
A faire
```

Rabin-Karp

Entrées : un texte T, un motif M
Sortie : la liste des positions de M dans T
Complexité : $\mathcal{O}(|T| + |M|)$ (max : $\mathcal{O}(|T| \times |M|)$)

```
RabinKarp (T, M) :
    L ← ∅
    n ← |T|
    m ← |M|
    hach_n ← Hachage(T[1..m])
    hach_m ← Hachage(M[1..m])
    Pour i allant de 0 a n-m+1 faire
        Si hach_n = hach_m alors
            Si T[i..i+m-1] = M alors
                L ← L ∪ {i}
    Retourner L
```

LZW

Entrées : un texte T
Sortie : la liste des codes de T
Complexité : $\mathcal{O}(|T|)$

```
LZWCompression (T) :
    L ← ∅
    dico ← ∅
    w ← ""
    Tant que il reste des caracteres a lire dans le texte, faire
        c ← prochain caractere
        p ← w · c
        Si p est dans le dico alors
            w ← p
        Sinon
            Ajouter p dans le dico
            L ← L ∪ dico(w)
            w ← c
    L ← L ∪ dico(w)
    Retourner L

LZWDecompression (Code, dico) :
    n ← |Code|
    L ← ∅
    v ← prochain(Code)
    L ← L ∪ v
    w ← char(v)

    Pour i allant de 2 a n faire
        k ← prochain(Code)
        Si k ∈ dico alors
            input ← dico(k)
        Sinon
            input ← w · w[0]
        L ← L ∪ input
        dico ← dico ∪ w · input[0]
        w ← input
    Retourner L
```

Algos de logique

Algorithme de Quine

Entrées : G une CNF, p une variable propositionnelle et $b \in \mathbb{B}$
Sortie : Une CNF équivalente à $G[p \mapsto b]$
Complexité : $\mathcal{O}(|G|)$

```
Assume (G, p, b) :
    Soit ℓ_V le litteral p si b = V, ¬p sinon
    Soit ℓ_F le litteral ¬p si b = V, p sinon
    Pour C ∈ G faire
        Si ℓ_V ∈ C alors
            Supprimer C de G
        Sinon
            Si ℓ_F ∈ C alors
            Supprimer ℓ_F de C
    Retourner G
```

Entrée : G une CNF
Sortie : Oui si G est satisfiable, Non sinon
Complexité : $\mathcal{O}(|G|)$

```
Quine (G) :
    Si G = ∅ alors
        Retourner Oui
    Sinon
        Si ∅ ∈ G alors
            Retourner Non
        Sinon Si ∃{ℓ} ∈ G alors
            Si ℓ = p avec p ∈ vars(G), alors
                Quine(Assume(G, p, V))
            Sinon Si ℓ = ¬p avec p ∈ vars(G), alors
                Quine(Assume(G, p, F))
        Sinon
            p ← h(G)
            On essaie Quine(Assume(G, p, V))
            On essaie Quine(Assume(G, p, F))
```

Algos des languages/automates

Berry-Sethi

Entrée : Une expression régulière e
Sortie : Un automate reconnaissant $\mathcal{L}(e)$
Complexité : $\mathcal{O}(|e|)$

```
On linearise e en f avec une fonction φ telle que f_φ = e
On calcule inductivement Λ(f), S(f), P(f), F(f)
On fabrique A = (Σ, Q, I, F, δ) un automate reconnaissant L(f)
Retourner A_φ
```

Algos d'apprentissage

K-moyennes

```
A faire
```

K plus proches voisins

Entrées : Un jeu de données classifiées (S, c), un vecteur d'entrée $\overline{v}$
Sortie : La classe de $\overline{v}$
Complexité : $\mathcal{O}(|S|)$

```
kNN (S, c, v̄) :
    On trie S par distance croissante de v̄ en d_1, d_2, ..., d_k, d_{k+1}, ...
    Soit D un dictionnaire de classes de C vers ℕ initialise a 0
    Pour i allant de 1 a k faire
        D[c(d_i)] ← D[c(d_i)] + 1
    Retourner la classe c maximisant D[c]
```

Arbres K-dimensionnels

Entrées : Un jeu de données $\mathcal{V}$ et $i \in [\![0, n-1]\!]$ où n est la dimension des données.
Sortie : Un arbre k-dimensionnel T
Complexité : $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}|)$

```
FabriqueArbreKD (V, i) :
    Si V = ∅ alors
        Retourner Vide
    Sinon
        On cherche v ∈ V tel que v_i est la mediane de {u_i | u ∈ V}
        Retourner Noeud((v, i), FabriqueArbreKD((V \ {v})^{≥_i v}, i + 1 mod n),
                                FabriqueArbreKD((V \ {v})^{≤_i v}, i + 1 mod n))
```

Entréés : Un arbre k-dimensionnel T, un vecteur v
Sortie : La classe de $\overline{v}$
Complexité : $\mathcal{O}(\log |\mathcal{V}|)$

```
RechercheKD (T, v̄) :
    Si T = Vide alors
        Retourner None
    Sinon
        Noeud((u, i), G, D) ← T
        Si u_i ≤ v_i alors
            W ← RechercheKD(D, v)
            Si W = None alors
                W' ← RechercheKD(G, v)
                Si W' = None alors
                    Retourner Some(u)
                Sinon
                    Some(z) ← W'
                    Retourner le plus proche de v entre u et z
            Sinon
                Some(w) ← W
                Si v_i − u_i ≤ d(v, w) alors
                    W' ← RechercheKD(G, v)
                    Si W' = None alors
                        Retourner Some(plus proche de v entre u et w)
                    Sinon
                        Some(z) ← W'
                        Retourner Some(plus proche de v entre u, w et z)
                Sinon
                    Retourner W
        Sinon Idem
```

Algorithme HAC

Entrées : un ensemble de points P
Sortie : un arbre de clustering
Complexité : $\mathcal{O}(|P|^2)$

```
HAC (P) :
    P ← {{x}|x ∈ 𝒟}
    Tant que |P| ≥ 2 et Crit, faire
        Soit A, B ∈ P minimisant D(A, B) avec A ≠ B
        P ← (P \ {A, B}) ∪ {A ∪ B}
    Retourner P
```

où $Crit$ est un critère sur le nombre de classes ou sur la valeur de la plus petite dissimilarité.

Algorithme ID3

```
    A faire
```