Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université de Paris

L2 Informatique & DL Bio-Info, Jap-Info, Math-Info

Année universitaire 2020-2021

dernier TP cette semaine (toujours avec une permanence sur discord jeudi matin et vendredi matin)

dernier TD la semaine prochaine ; groupes INFO1 et INFO2 sur discord (probablement mardi 11 à 8h30)

contrôle n° 3 mercredi prochain (12 mai) sur moodle, de 15h à 16h30

Quelques applications des tris

Applications du tri en géométrie : 1. Calcul de l'enveloppe convexe

enveloppe convexe d'une partie $\mathcal{P}$ du plan : plus petite partie convexe $\mathcal{C}$ contenant $\mathcal{P}$

enveloppe convexe d'une partie $\mathcal{P}$ du plan : plus petite partie convexe $\mathcal{C}$ contenant $\mathcal{P}$

si $\mathcal{P}$ est un ensemble fini de points (on parle de *nuage* de points),
$\mathcal{C}$ est un polygone dont les sommets sont des points du nuage

Applications du tri en géométrie : 1. Calcul de l'enveloppe convexe

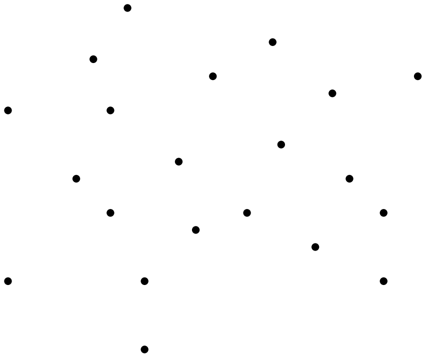
enveloppe convexe d'une partie $\mathcal{P}$ du plan : plus petite partie convexe $\mathcal{C}$ contenant $\mathcal{P}$

si $\mathcal{P}$ est un ensemble fini de points (on parle de *nuage* de points), $\mathcal{C}$ est un polygone dont les sommets sont des points du nuage

enveloppe_convexe(nuage)

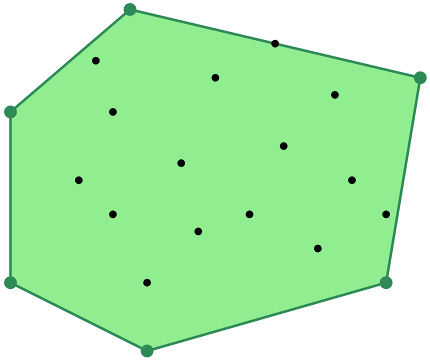
étant donné un `nuage` de points du plan, déterminer l'enveloppe convexe des points du `nuage`

Caractérisation des arêtes de l'enveloppe



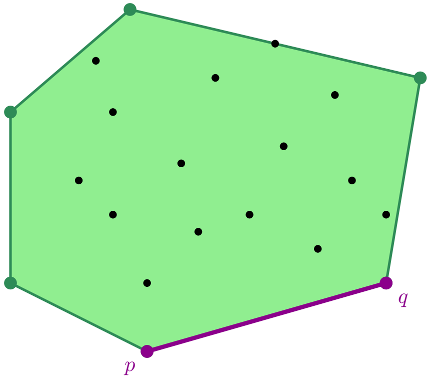
Un nuage de points

Caractérisation des arêtes de l'enveloppe



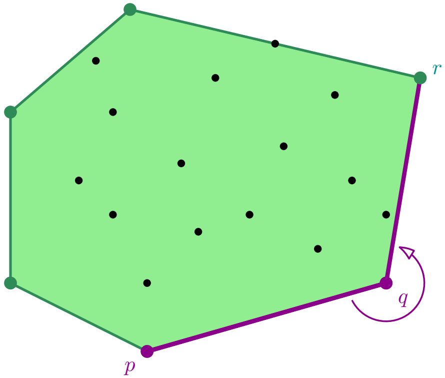
Son enveloppe convexe

Caractérisation des arêtes de l'enveloppe



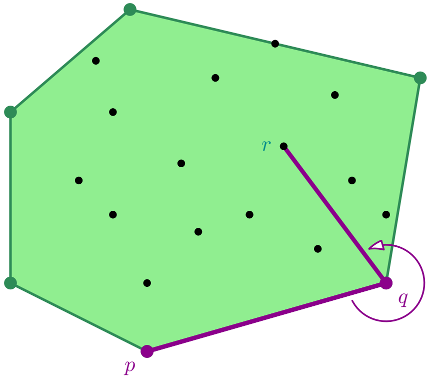
Une arête $[pq]$ de l'enveloppe (dans le sens direct)

Caractérisation des arêtes de l'enveloppe



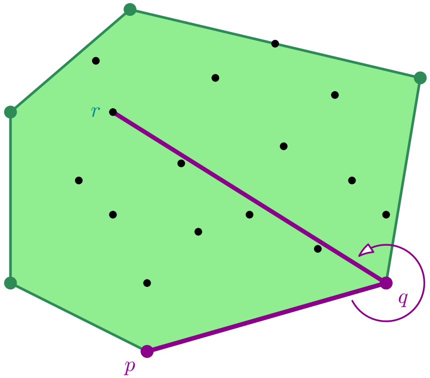
Tous les angles $\widehat{pqr}$ « tournent à gauche »

Caractérisation des arêtes de l'enveloppe



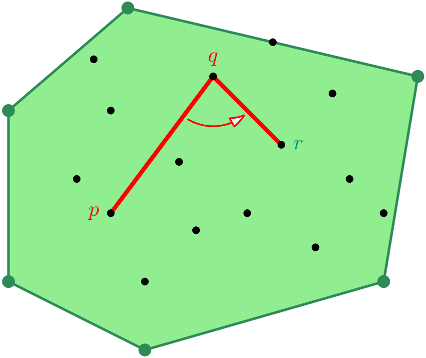
Tous les angles $\widehat{pqr}$ « tournent à gauche »

Caractérisation des arêtes de l'enveloppe



Tous les angles $\widehat{pqr}$ « tournent à gauche »

Caractérisation des arêtes de l'enveloppe



... contrairement au cas où [pq] n'est pas une arête de l'enveloppe

Enveloppe convexe d'un nuage – méthode naïve

```
def enveloppe_convexe_naive(nuage) :
  tous_les_couples =     # tous les couples de points du nuage
          [ (p,q) for p in nuage for q in nuage if p != q ]
  aretes_enveloppe = []
  for (p, q) in tous_les_couples :
    for r in nuage : # r contredit-il la caractérisation pour [pq]?
      if tourne_a_droite(p, q, r) : break
    else : # ie si la boucle termine normalement, [pq] ∈ enveloppe
      aretes_enveloppe += [(p,q)]
  return aretes_enveloppe
```

Enveloppe convexe d'un nuage – méthode naïve

```
def enveloppe_convexe_naive(nuage) :
  tous_les_couples =     # tous les couples de points du nuage
        [ (p,q) for p in nuage for q in nuage if p != q ]
  aretes_enveloppe = []
  for (p, q) in tous_les_couples :
    for r in nuage : # r contredit-il la caractérisation pour [pq]?
      if tourne_a_droite(p, q, r) : break
    else : # ie si la boucle termine normalement, [pq] ∈ enveloppe
      aretes_enveloppe += [(p,q)]
  return aretes_enveloppe
```

Lemme

enveloppe_convexe_naive(nuage) retourne une liste formée des arêtes de l'enveloppe convexe de nuage en temps $\Theta(n^3)$ (au pire et en moyenne)

Idée : pour être plus efficace, il ne faut pas considérer *tous* les couples mais essayer de « tourner » autour du nuage

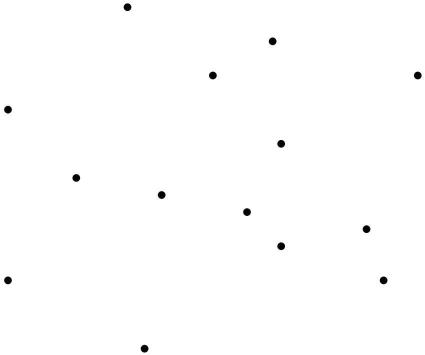
Plus précisément :

- partir d'un point « extrémal » p_0 – par exemple celui d'ordonnée minimale – qui appartient nécessairement à l'enveloppe
- considérer ensuite les points un par un – p_1, p_2,... p_{n-1} pour déterminer si p_i appartient à l'enveloppe convexe du nuage $\{p_0, p_1, \ldots, p_i\}$

Question : dans quel ordre faut-il considérer les points du nuage ?

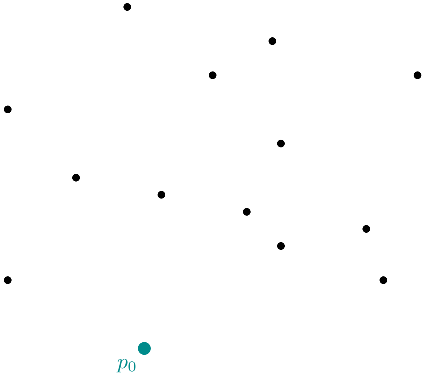
Enveloppe convexe d'un nuage – par balayage

Exemple :



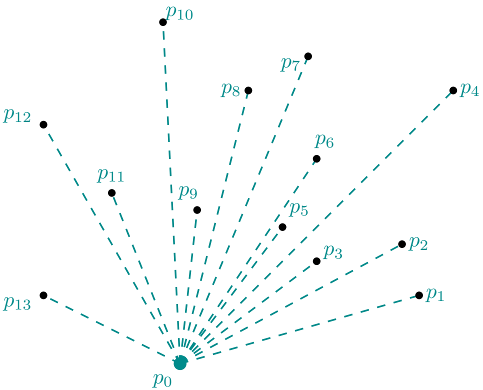
Enveloppe convexe d'un nuage – par balayage

Exemple :



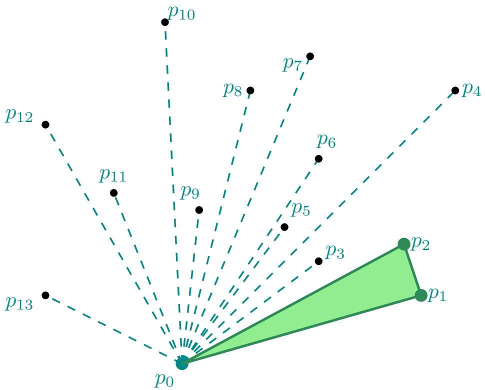
Enveloppe convexe d'un nuage – par balayage

Exemple :



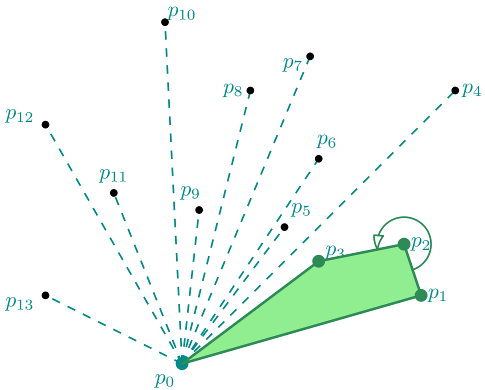
Enveloppe convexe d'un nuage – par balayage

Exemple :



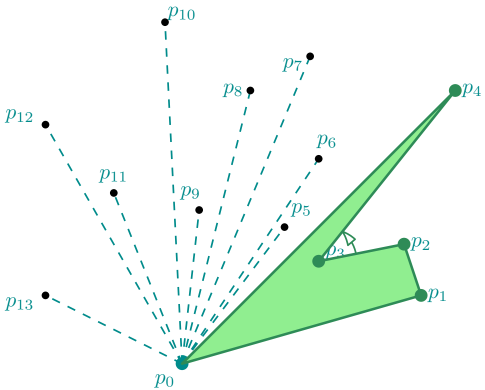
Enveloppe convexe d'un nuage – par balayage

Exemple :



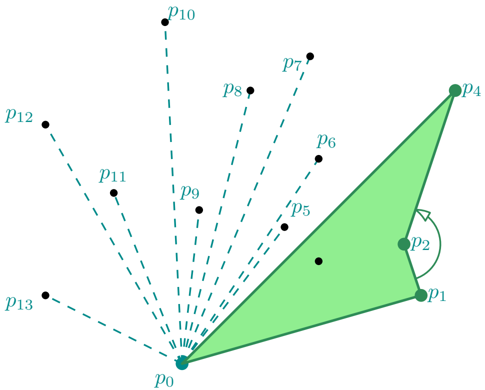
Enveloppe convexe d'un nuage – par balayage

Exemple :



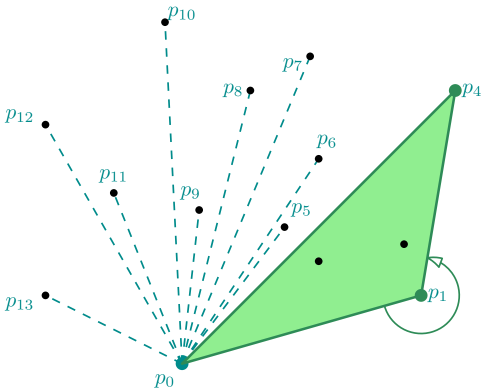
Enveloppe convexe d'un nuage – par balayage

Exemple :



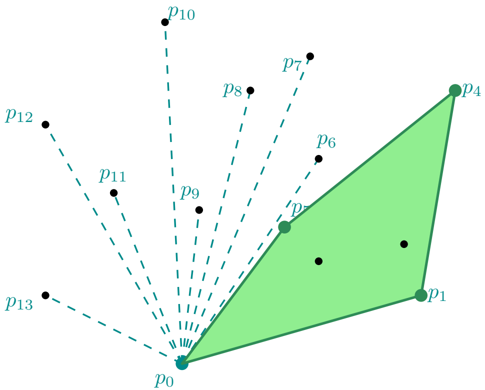
Enveloppe convexe d'un nuage – par balayage

Exemple :



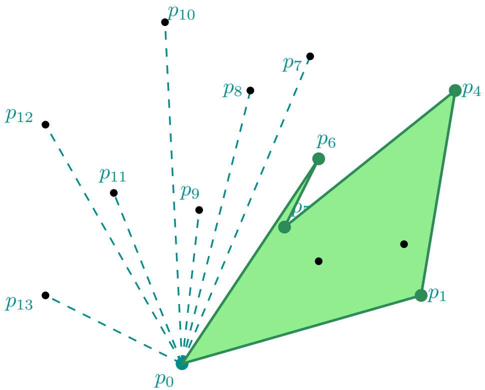
Enveloppe convexe d'un nuage – par balayage

Exemple :

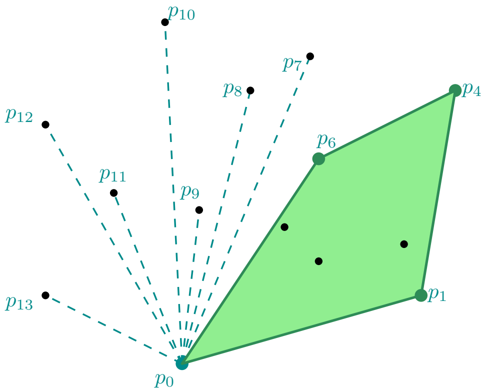


Enveloppe convexe d'un nuage – par balayage

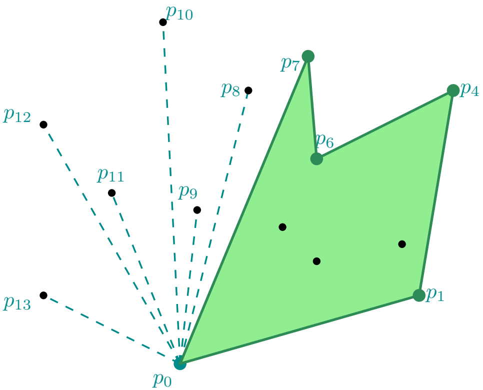
Exemple :



Exemple :

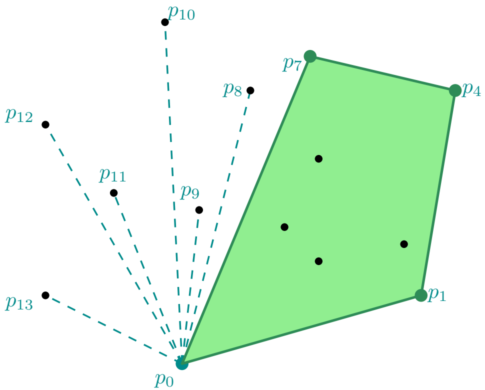


Exemple :



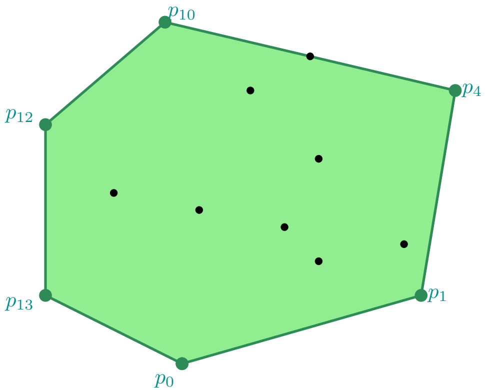
Enveloppe convexe d'un nuage – par balayage

Exemple :



Enveloppe convexe d'un nuage – par balayage

Exemple :



Enveloppe convexe d'un nuage – par balayage

```
def enveloppe_convexe_par_balayage(nuage) :
  p0 = point_le_plus_bas(nuage)
  angles = [ angle_polaire(point, p0) for point in nuage ]
  nuage_trié = trier_selon_angles(nuage, angles)
  pile = [ nuage_trié[0], nuage_trié[1], nuage_trié[2] ]
  for point in nuage_trié :
    while tourne_a_droite(pile[-2], pile[-1], point) :
      pile.pop()
    pile.append(point)
  return pile
```

Enveloppe convexe d'un nuage – par balayage

```
def enveloppe_convexe_par_balayage(nuage) :
  p0 = point_le_plus_bas(nuage)
  angles = [ angle_polaire(point, p0) for point in nuage ]
  nuage_trié = trier_selon_angles(nuage, angles)
  pile = [ nuage_trié[0], nuage_trié[1], nuage_trié[2] ]
  for point in nuage_trié :
    while tourne_a_droite(pile[-2], pile[-1], point) :
      pile.pop()
    pile.append(point)
  return pile

def trier_selon_angles(nuage, angles) :
  # exemple de « decorate-sort-undecorate »
  return [ point
        for (angle, point) in sorted(zip(angles, nuage)) ]
```

Théorème

`enveloppe_convexe_par_balayage(nuage)` produit la liste des sommets de l'enveloppe convexe en temps $\Theta(n \log n)$

Démonstration

- point le plus bas : c'est juste un `min` $\Longrightarrow \Theta(n)$
- tri selon l'angle : $\Theta(n \log n)$
- double boucle : $\Theta(n)$ car chacun des n points est, au pire, sorti une fois de la pile

points_les_plus_proches(nuage)

étant donné un nuage de points du plan, déterminer les deux points du nuage les plus proches l'un de l'autre

points_les_plus_proches(nuage)

étant donné un nuage de points du plan, déterminer les deux points du nuage les plus proches l'un de l'autre

problème presque équivalent :

distance_minimale(nuage)

étant donné un nuage de points du plan, déterminer la distance minimale entre deux points du nuage

Cette distance minimale est appelée *maille* du nuage de points

distance_minimale(nuage)

étant donné un nuage de points du plan, déterminer la distance minimale entre deux éléments du nuage

```
def distance_minimale_naive(nuage) :
  toutes_les_distances =
     [ distance(p,q) for p in nuage for q in nuage if p != q ]
  return min(toutes_les_distances)
```

distance_minimale(nuage)

étant donné un nuage de points du plan, déterminer la distance minimale entre deux éléments du nuage

```
def distance_minimale_naive(nuage) :
  toutes_les_distances =
     [ distance(p,q) for p in nuage for q in nuage if p != q ]
  return min(toutes_les_distances)
```

Lemme

distance_minimale_naive(nuage) calcule la distance minimale entre deux points du nuage en temps $\Theta(n^2)$

distance_minimale(nuage)

étant donné un nuage de points du plan, déterminer la distance minimale entre deux éléments du nuage

Approche ***« diviser pour régner »*** :

- découper le problème en sous-problèmes de taille inférieure
- résoudre ***récursivement*** le ou les sous-problèmes
- résoudre le problème initial à l'aide des résultats des sous-problèmes

distance_minimale(nuage)

étant donné un `nuage` de points du plan, déterminer la distance minimale entre deux éléments du `nuage`

Approche ***« diviser pour régner »*** :

- séparer `nuage` en deux sous-listes `gauche` et `droite`
- résoudre ***récursivement*** le ou les sous-problèmes
- résoudre le problème initial à l'aide des résultats des sous-problèmes

distance_minimale(nuage)

étant donné un `nuage` de points du plan, déterminer la distance minimale entre deux éléments du `nuage`

Approche ***« diviser pour régner »*** :

- séparer `nuage` en deux sous-listes `gauche` et `droite`
- calculer `d1 = distance_minimale(gauche)`
  et `d2 = distance_minimale(droite)`
- résoudre le problème initial à l'aide des résultats des sous-problèmes

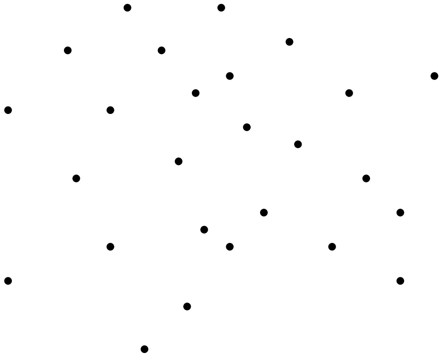
distance_minimale(nuage)

étant donné un `nuage` de points du plan, déterminer la distance minimale entre deux éléments du `nuage`

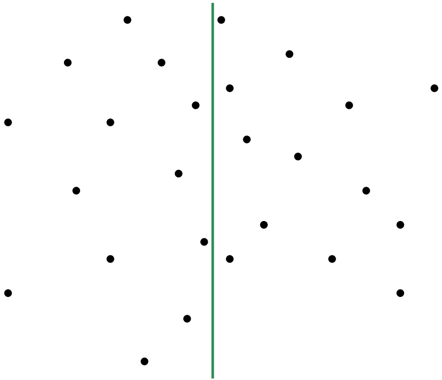
Approche « ***diviser pour régner*** » :

- séparer `nuage` en deux sous-listes `gauche` et `droite`
- calculer `d1 = distance_minimale(gauche)`
  et `d2 = distance_minimale(droite)`
- chercher s'il existe `p1` dans `gauche` et `p2` dans `droite` plus proches que `min(d1, d2)`

Points les plus proches – *« diviser pour régner »*

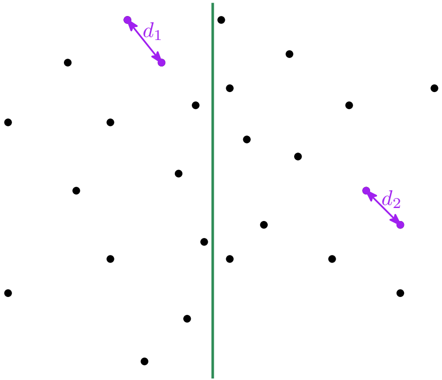


Points les plus proches – « *diviser pour régner* »



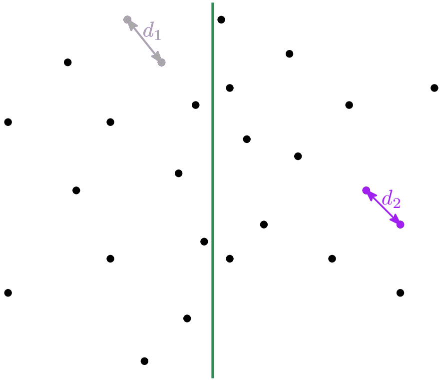
Partitionnement gauche – droite

Points les plus proches – « *diviser pour régner* »



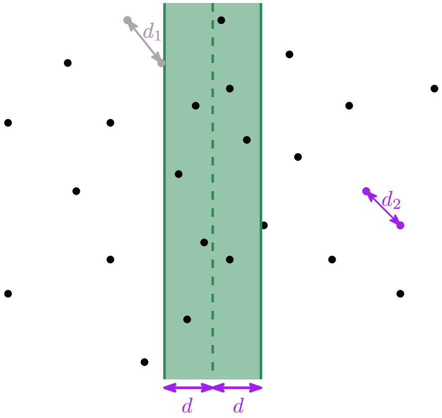
Appels récursifs sur `gauche` et `droite`

Points les plus proches – *« diviser pour régner »*



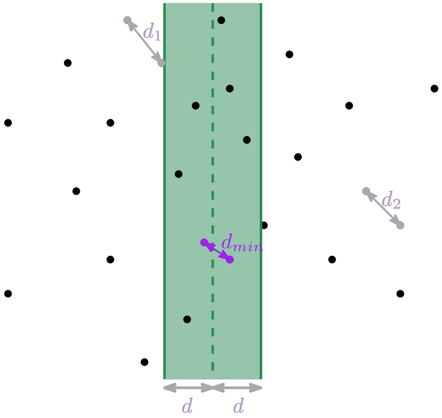
Calcul de $d = \min(d1, d2)$

Points les plus proches – « *diviser pour régner* »



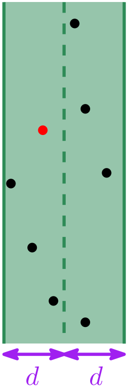
Extraction de la bande médiane de largeur 2d

Points les plus proches – « *diviser pour régner* »

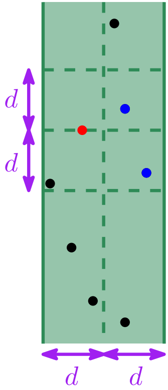


Recherche dans la bande médiane

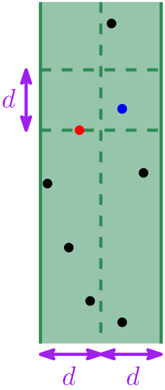
Comment trouver (`p1`, `p2`) efficacement ?



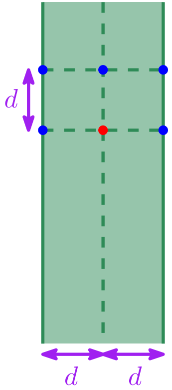
Comment trouver (`p1`, `p2`) efficacement ?



Comment trouver (`p1`, `p2`) efficacement ?



Comment trouver (`p1`, `p2`) efficacement ?



Comment optimiser l'algorithme ?

Pour le partitionnement gauche–droite

Trier ***une fois pour toutes*** la liste des points selon les abscisses
$\Longrightarrow$ étant donné `L_x`, le partitionnement a un coût constant

Comment optimiser l'algorithme ?

Pour le partitionnement gauche–droite

Trier *une fois pour toutes* la liste des points selon les abscisses
$\Longrightarrow$ étant donné `L_x`, le partitionnement a un coût constant

Pour la recherche des couples (p1, p2)

Trier *une fois pour toutes* la liste des points selon les ordonnées
$\Longrightarrow$ étant donné `L_y`, la recherche a un coût linéaire

Comment optimiser l'algorithme ?

Pour le partitionnement gauche–droite

Trier ***une fois pour toutes*** la liste des points selon les abscisses
$\Longrightarrow$ étant donné `L_x`, le partitionnement a un coût constant

Pour la recherche des couples (p1, p2)

Trier ***une fois pour toutes*** la liste des points selon les ordonnées
$\Longrightarrow$ étant donné `L_y`, la recherche a un coût linéaire

$$C_{totale}(n) = C_{tris}(n) + C_{rec}(n) = \Theta(n \log n) + C_{rec}(n)$$

$$C_{rec}(n) = 2C_{rec}\left(\frac{n}{2}\right) + O(n)$$

$$\Longrightarrow \quad C_{totale}(n) \in \Theta(n \log n)$$

Étant donné un (petit) `motif` et un (corpus de) (long(s)) `texte`(s), déterminer si `motif` apparaît dans `texte`

Étant donné un (petit) `motif` et un (corpus de) (long(s)) `texte`(s), déterminer si `motif` apparaît dans `texte`

botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin aiguille botte de foin aiguille botte de foin aiguille botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin ...

Étant donné un (petit) `motif` et un (corpus de) (long(s)) `texte`(s), déterminer si `motif` apparaît dans `texte`

botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin aiguille *botte de foin aiguille botte de foin aiguille botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin . . .*

Application en algorithmique du texte : Recherche de motif dans un texte *(variante)*

Étant donné un (petit) `motif` et un (corpus de) (long(s)) `texte`(s), déterminer ***toutes les occurrences*** de `motif` dans `texte`

*botte de foin botte de foin botte de foin botte de foin botte de foin botte
de foin botte de foin botte de foin botte de foin botte de foin botte de foin
botte de foin botte de foin botte de foin botte de foin botte de foin botte
de foin botte de foin* aiguille *botte de foin botte de foin botte de foin botte
de foin botte de foin botte de foin botte de foin botte de foin botte de foin
botte de foin botte de foin botte de foin botte de foin botte de foin botte
de foin botte de foin botte de foin botte de foin botte de foin* aiguille *botte
de foin botte de foin botte de foin botte de foin botte de foin botte de foin
botte de foin botte de foin botte de foin botte de foin botte de foin botte
de foin botte de foin botte de foin botte de foin botte de foin botte de foin
botte de foin botte de foin botte de foin botte de foin botte de foin botte
de foin botte de foin botte de foin botte de foin* aiguille *botte de foin botte
de foin botte de foin botte de foin botte de foin botte de foin botte de foin
botte de foin . . .*

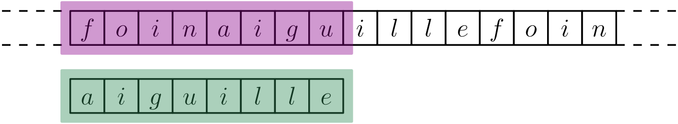
Étant donné un (petit) `motif` et un (corpus de) (long(s)) `texte`(s), compter ***toutes les occurrences*** de `motif` dans `texte`

botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin aiguille *botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin* aiguille *botte de foin botte de foin* aiguille *botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin . . .*

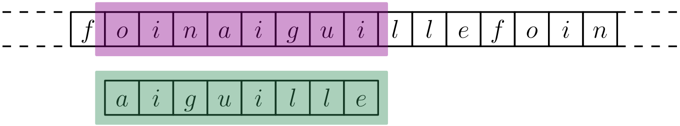
Algorithme naïf

f	o	i	n	a	i	g	u	i	l	l	e	f	o	i	n

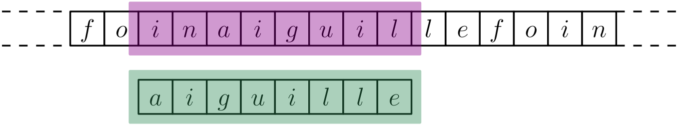
a	i	g	u	i	l	l	e

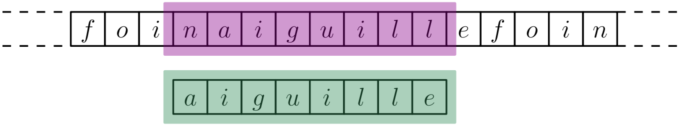

f o i n a i g u i l l e f o i n
a i g u i l l e

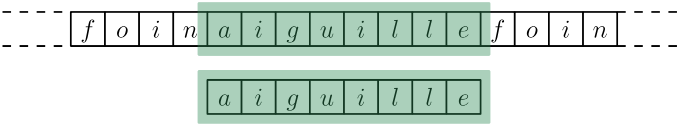
Algorithme naïf



Algorithme naïf




f o i n a i g u i l l e f o i n
a i g u i l l e


f o i n a i g u i l l e f o i n
a i g u i l l e

```
def recherche_naive(motif, texte) :
  for fenetre in facteurs(texte, len(motif)) :
    if fenetre == motif :
      return True
  return False
```

où `facteurs(texte, longueur)` est un générateur permettant d'itérer sur les facteurs de `texte` de `longueur` fixée :

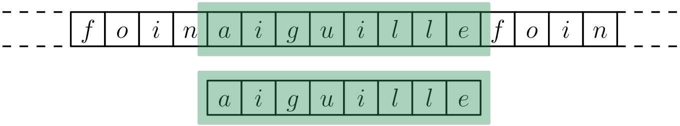
```
def facteurs(texte, longueur) :
  n = len(texte) - longueur + 1
  for i in range(n) :
    yield texte[i:i+longueur]
```

```
def comptage_naif(motif, texte) :
  cpt = 0
  for fenetre in facteurs(texte, len(motif)) :
    if fenetre == motif :
      cpt += 1
  return cpt
```

où `facteurs(texte, longueur)` est un générateur permettant d'itérer sur les facteurs de `texte` de `longueur` fixée :

```
def facteurs(texte, longueur) :
  n = len(texte) - longueur + 1
  for i in range(n) :
    yield texte[i:i+longueur]
```

Algorithme naïf



la complexité de cet algorithme est en $\Theta(mn)$ pour un `motif` de longueur m et `texte` de longueur n

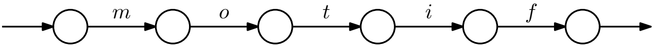
peut-on faire mieux ?

le choix du meilleur algorithme dépend de plusieurs facteurs :

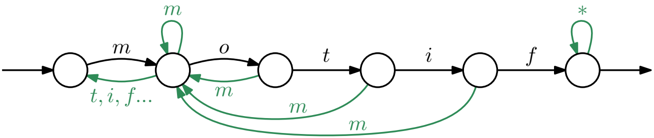
- un ou plusieurs textes ?
- connu(s) à l'avance ou traité(s) *online* ?
- un ou plusieurs motifs ?
- vraiment petits ou pas ?
- taille de l'alphabet ?

Cas de recherches répétées d'un motif

Réponse standard : avec un automate

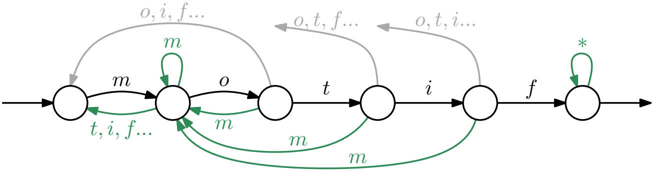


Réponse standard : avec un automate



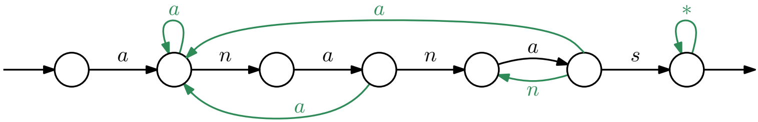
Cas de recherches répétées d'un motif

Réponse standard : avec un automate



Cas de recherches répétées d'un motif

Réponse standard : avec un automate

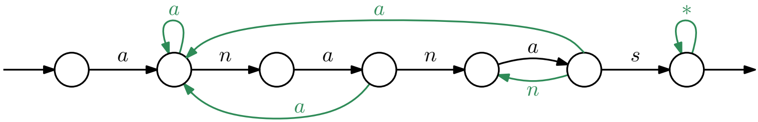


avantages :

- une fois l'automate construit, complexité en $\Theta(n)$
- algorithme pouvant fonctionner ***online***

Cas de recherches répétées d'un motif

Réponse standard : avec un automate



avantages :

- une fois l'automate construit, complexité en $\Theta(n)$
- algorithme pouvant fonctionner ***online***

inconvénients : (construction et) stockage de l'automate
si l'alphabet a k lettres, en version naïve (mais améliorable) :

- complexité en temps $O(m^3k)$
- complexité en espace $O(mk)$

contrainte : ne (presque) rien stocker

(*cf* TD)

Contexte : long `texte` connu à l'avance, de longueur n

Contexte : long `texte` connu à l'avance, de longueur n

Cahier des charges : moyennant un prétraitement éventuel de `texte` (*indexation*), pouvoir faire ensuite des recherches de motifs en temps **sous-linéaire** en n

Contexte : long `texte` connu à l'avance, de longueur n

Cahier des charges : moyennant un prétraitement éventuel de `texte` (*indexation*), pouvoir faire ensuite des recherches de motifs en temps **sous-linéaire** en n

(Une) solution : construire la *table des suffixes* du `texte`

Contexte : long `texte` connu à l'avance, de longueur n

Cahier des charges : moyennant un prétraitement éventuel de `texte` (*indexation*), pouvoir faire ensuite des recherches de motifs en temps **sous-linéaire** en n

(Une) solution : construire la *table des suffixes* du `texte`

Définition

• suffixe d'indice i de `texte` : facteur (sous-tableau) `texte[i:]`

• table des suffixes de `texte` : tableau ordonné des (indices des) suffixes selon l'ordre ***lexicographique*** (*i.e.* alphabétique) :

$i \prec j \iff$ `texte[i:]` précède `texte[j:]` lexicographiquement.

Table des suffixes

```
>>> texte = "quelbonbonbon"
>>> suffixes = [ (texte[i:], i) for i in range(len(texte)) ]
>>> suffixes
[('quelbonbonbon', 0), ('uelbonbonbon', 1), ('elbonbonbon', 2),
('lbonbonbon', 3), ('bonbonbon', 4), ('onbonbon', 5),
('nbonbon', 6), ('bonbon', 7), ('onbon', 8), ('nbon', 9),
('bon', 10), ('on', 11), ('n', 12)]
>>> sorted(suffixes)
[('bon', 10), ('bonbon', 7), ('bonbonbon', 4),
('elbonbonbon', 2), ('lbonbonbon', 3), ('n', 12), ('nbon', 9),
('nbonbon', 6), ('on', 11), ('onbon', 8), ('onbonbon', 5),
('quelbonbonbon', 0), ('uelbonbonbon', 1)]
>>> table_suffixes = [ i for (suff, i) in sorted(suffixes) ]
>>> table_suffixes
[10, 7, 4, 2, 3, 12, 9, 6, 11, 8, 5, 0, 1]
```

notations : `texte` de longueur n, `motif` de longueur m

Exploitation de la table des suffixes

notations : `texte` de longueur n, `motif` de longueur m

comparaison (lexicographique) de 2 mots de longueurs ℓ_1 et ℓ_2 :

$\Longrightarrow$ temps $\Theta(\min(\ell_1, \ell_2))$

Exploitation de la table des suffixes

notations : `texte` de longueur n, `motif` de longueur m

comparaison (lexicographique) de 2 mots de longueurs ℓ_1 et ℓ_2 :

$\Longrightarrow$ temps $\Theta(\min(\ell_1, \ell_2))$

recherche d'un motif : ***recherche dichotomique***

$\Longrightarrow$ $\Theta(\log n)$ comparaisons de mots

$\Longrightarrow$ temps $\Theta(m \log n)$ (au pire)

Exploitation de la table des suffixes

notations : texte de longueur n, motif de longueur m

comparaison (lexicographique) de 2 mots de longueurs ℓ_1 et ℓ_2 :

$\Longrightarrow$ temps $\Theta(\min(\ell_1, \ell_2))$

recherche d'un motif : ***recherche dichotomique***

$\Longrightarrow$ $\Theta(\log n)$ comparaisons de mots

$\Longrightarrow$ temps $\Theta(m \log n)$ (au pire)

recherche de toutes les occurrences d'un motif :

temps $\Theta(m \log n)$

Exploitation de la table des suffixes

notations : `texte` de longueur n, `motif` de longueur m

comparaison (lexicographique) de 2 mots de longueurs ℓ_1 et ℓ_2 :

$\Longrightarrow$ temps $\Theta(\min(\ell_1, \ell_2))$

recherche d'un motif : ***recherche dichotomique***

$\Longrightarrow$ $\Theta(\log n)$ comparaisons de mots

$\Longrightarrow$ temps $\Theta(m \log n)$ (au pire)

recherche de toutes les occurrences d'un motif :

temps $\Theta(m \log n)$

recherche du motif de longueur k le plus fréquent :

temps $\Theta(kn)$, espace $\Theta(k)$

Construction naïve : par un tri fusion, en $\Theta(n \log n)$ comparaisons ***entre suffixes***, donc en temps $O(n^2 \log n)$

Construction naïve : par un tri fusion, en $\Theta(n \log n)$ comparaisons ***entre suffixes***, donc en temps $O(n^2 \log n)$

Pourtant :

Théorème

la table des suffixes d'un texte de longueur n *peut être construite en temps* $\Theta(n)$

Construction de la table des suffixes

Construction naïve : par un tri fusion, en $\Theta(n \log n)$ comparaisons ***entre suffixes***, donc en temps $O(n^2 \log n)$

Pourtant :

Théorème

la table des suffixes d'un texte de longueur n peut être construite en temps $\Theta(n)$

Outils :

Tri par base : permet de trier lexicographiquement n mots de ***longueur fixée*** sur un ***alphabet fini*** en temps $\Theta(n)$

Philosophie « diviser pour régner » : construction (et tri) d'un sous-ensemble de ***suffixes-échantillons*** puis étape de « fusion »

Construction de la table des suffixes

1. ajouter à l'alphabet A une lettre `'0'`, inférieure à toutes les autres lettres, et compléter `texte` avec 0, 1 ou 2 caractères `'0'` pour que $n = \texttt{len}(\texttt{texte}) \equiv 0 \mod 3$
2. soit S l'ensemble des suffixes de `texte` ; partitionner S en $S_0 \sqcup S_1 \sqcup S_2$, où $S_i = \{\texttt{texte[j:]} \mid j \equiv i \mod 3\}$
3. soit $E = S_1 \sqcup S_2$ l'ensemble des *suffixes-échantillons* de `texte` ; définir un texte `aux` dont l'ensemble des suffixes correspond à E par une bijection respectant l'ordre
4. calculer récursivement la table des suffixes de `aux`, et trier E en conséquence
5. trier S_0
6. fusionner S_0 et E

Texte auxiliaire

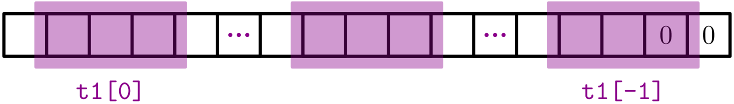
1. ajouter encore 2 caractères `'0'` à `texte`
2. définir `t1` et `t2` : mots sur l'alphabet A^3 dont la « projection » sur A^* est respectivement `texte[1:-1]` et `texte[2:]` :

															0	0

3. trier en temps linéaire les éléments de `t1 + t2`, donc déterminer le rang de chaque « lettre-triplet » apparaissant dans l'un des deux mots ;
4. `aux` est le mot obtenu à partir de `t1 + t2` en substituant son rang à chaque lettre-triplet.

Texte auxiliaire

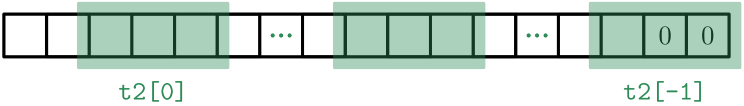
1. ajouter encore 2 caractères `'0'` à `texte`
2. définir `t1` et `t2` : mots sur l'alphabet A^3 dont la « projection » sur A^* est respectivement `texte[1:-1]` et `texte[2:]` :



3. trier en temps linéaire les éléments de `t1 + t2`, donc déterminer le rang de chaque « lettre-triplet » apparaissant dans l'un des deux mots ;
4. `aux` est le mot obtenu à partir de `t1 + t2` en substituant son rang à chaque lettre-triplet.

Texte auxiliaire

1. ajouter encore 2 caractères `'0'` à `texte`
2. définir `t1` et `t2` : mots sur l'alphabet A^3 dont la « projection » sur A^* est respectivement `texte[1:-1]` et `texte[2:]` :



3. trier en temps linéaire les éléments de `t1 + t2`, donc déterminer le rang de chaque « lettre-triplet » apparaissant dans l'un des deux mots ;
4. `aux` est le mot obtenu à partir de `t1 + t2` en substituant son rang à chaque lettre-triplet.

Exemple : `texte = "quelbonbonbon"`

```
t1 = ["uel", "bon", "bon", "bon", "000"]
t2 = ["elb", "onb", "onb", "on0", "000"]
```

ordre des triplets : $(000) < (bon) < (elb) < (on0) < (onb) < (uel)$
donc si on les numérote de 1 à 6 :

$$\mathtt{aux} = 6\ 2\ 2\ 2\ 1\ 3\ 5\ 5\ 4\ 1$$

Étant donné un alphabet A de taille ℓ ***fini*** et un entier k, trier une liste `L` d'éléments de A^k selon l'ordre lexicographique (*i.e.* alphabétique) le plus efficacement possible

```python
def tri_par_base(L, A, k) :
  tmp = L
  # numérotation des lettres selon l'ordre alphabétique
  dico = { lettre : i for (i, lettre) in enumerate(A) }
  # tri selon chaque position, en partant de la dernière
  for i in range(k) :
    aux = [ [] for lettre in A ]
    for mot in tmp :
      aux[dico[mot[k-1-i]]].append(mot)
    tmp = []
    for liste in aux : tmp += liste
  return tmp
```

Lemme

la i^e *étape du tri par base réalise un tri* ***stable*** *des mots de* L *selon la lettre de position* $k - i$

Lemme

chaque étape est de complexité $\Theta(n + \ell)$, *où* n *est la longueur de* L, *et* ℓ *la taille de l'alphabet – donc* $\Theta(n)$ *si l'alphabet est fixé une fois pour toute*

Théorème

le tri par base réalise le tri lexicographique de n *mots de longueur* k *en temps* $\Theta(nk)$