Module EA4 – Éléments d'Algorithmique

Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique
Année universitaire 2014-2015

Contrôle continu

Interrogation n° 2 à la place de l'amphi mercredi 1er avril

Interrogation n° 3 : (très probablement) TP noté les 5 et 6 mai

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

enveloppe_convexe(L)

étant donné une liste L de points du plan, déterminer l'enveloppe convexe des éléments de L

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

enveloppe_convexe(L)

étant donné une liste L de points du plan, déterminer l'enveloppe convexe des éléments de L

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

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

enveloppe_convexe(L)

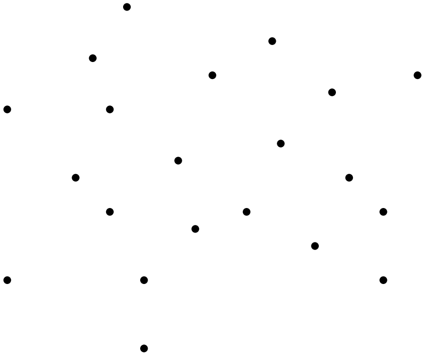
étant donné une liste L de points du plan, déterminer l'enveloppe convexe des éléments de L

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, $\mathcal{C}$ est un polygone dont les sommets sont des éléments de $\mathcal{P}$

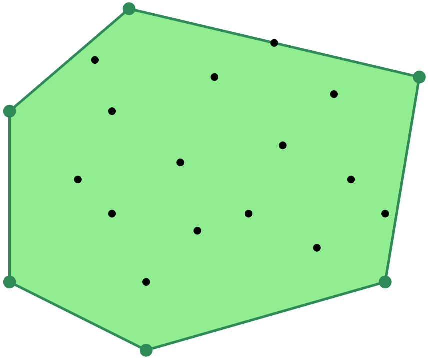
Enveloppe convexe de points

Exemple :



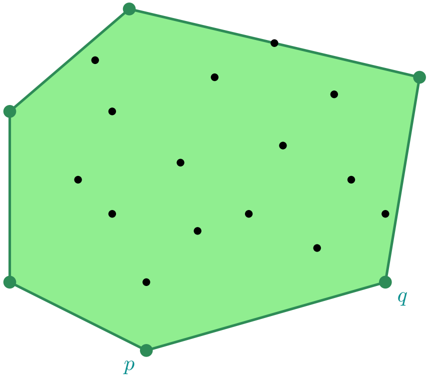
Enveloppe convexe de points

Exemple :



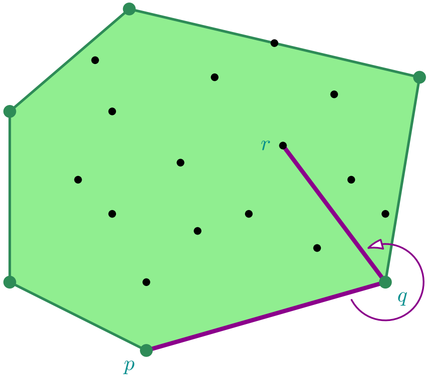
Enveloppe convexe de points

Exemple :



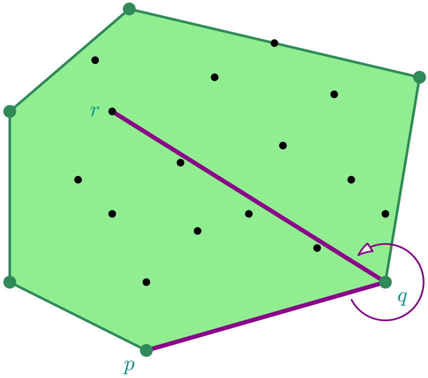
Enveloppe convexe de points

Exemple :



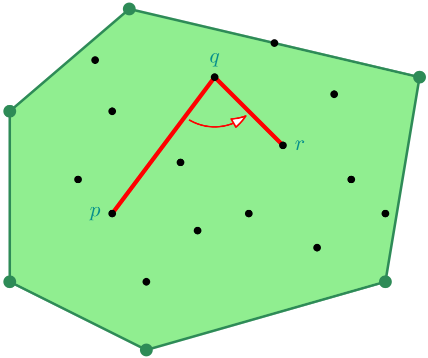
Enveloppe convexe de points

Exemple :



Enveloppe convexe de points

Exemple :



Enveloppe convexe de points

```
def enveloppe_convexe_naive(L) :
  couples = [ (p,q) for p in L for q in L if p != q ]
  aretes = []
  for (p, q) in couples :
    for r in L :
      if tourne a_droite(p, q, r) : break
    else : # ssi la boucle termine normalement
      aretes += [(p,q)]
  return aretes
```

Enveloppe convexe de points

```
def enveloppe_convexe_naive(L) :
  couples = [ (p,q) for p in L for q in L if p != q ]
  aretes = []
  for (p, q) in couples :
    for r in L :
      if tourne a_droite(p, q, r) : break
    else : # ssi la boucle termine normalement
      aretes += [(p,q)]
  return aretes
```

Lemme

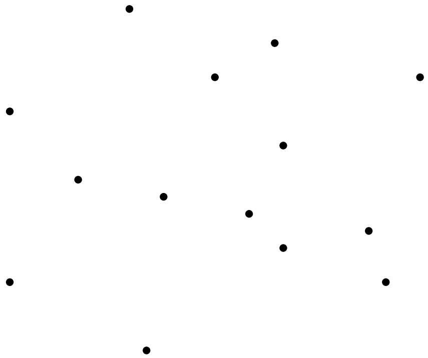
enveloppe_convexe_naive(L) retourne une liste formée des arêtes de l'enveloppe convexe de L en temps $\Theta(n^3)$ *(au pire et en moyenne)*

Enveloppe convexe de points

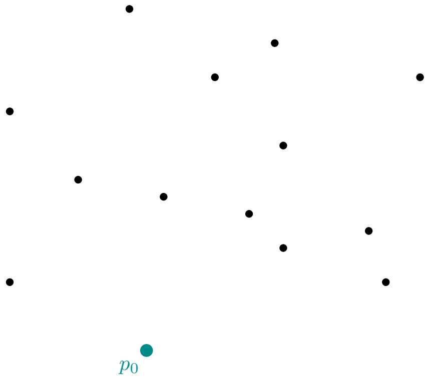
```
def enveloppe_convexe_par_balayage(L) :
  p0 = point_le_plus_bas(L)
  L = trier_selon_angle_polaire(L, p0)
  pile = [L[0], L[1], L[2]]
  for point in L :
    while tourne_a_droite(pile[-2], pile[-1], point) :
      pile.pop()
    pile.append(point)
  return pile
```

Enveloppe convexe de points

Exemple :

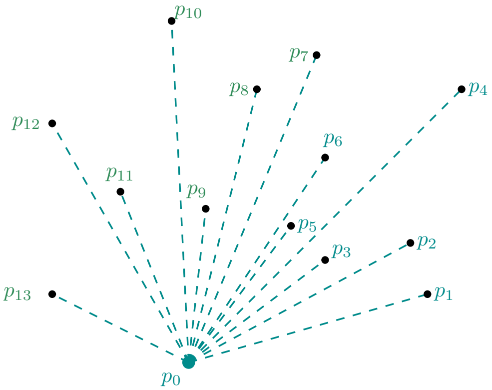


Exemple :



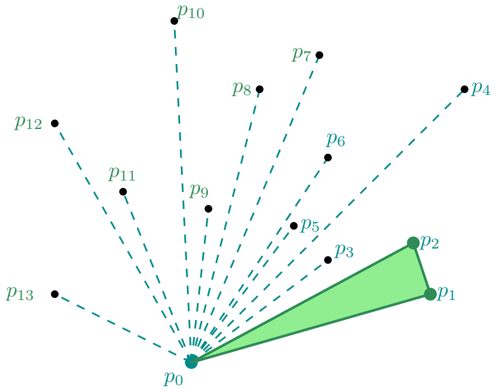
Enveloppe convexe de points

Exemple :



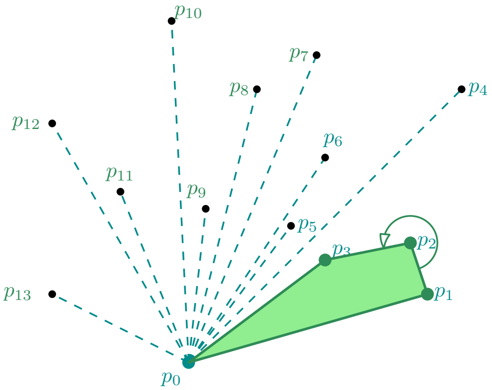
Enveloppe convexe de points

Exemple :



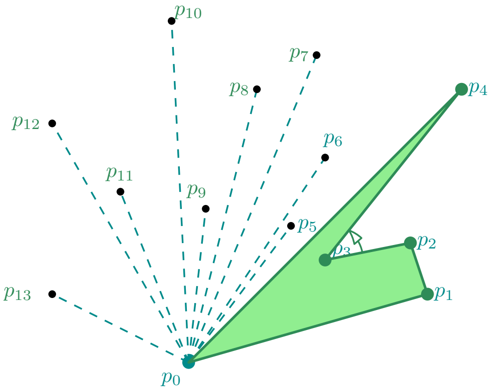
Enveloppe convexe de points

Exemple :



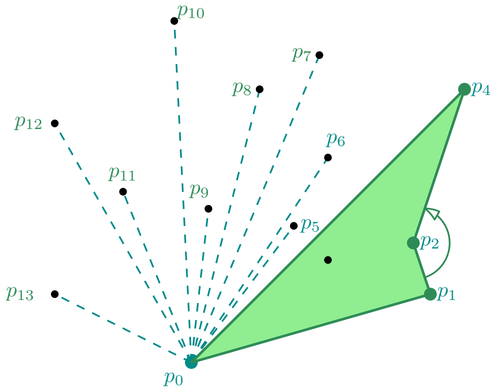
Enveloppe convexe de points

Exemple :

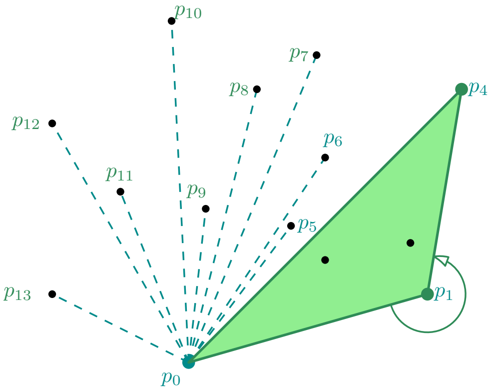


Enveloppe convexe de points

Exemple :

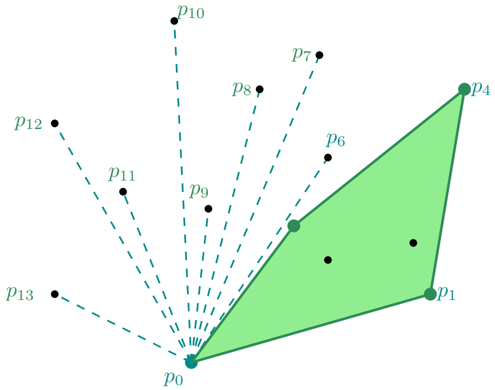


Exemple :



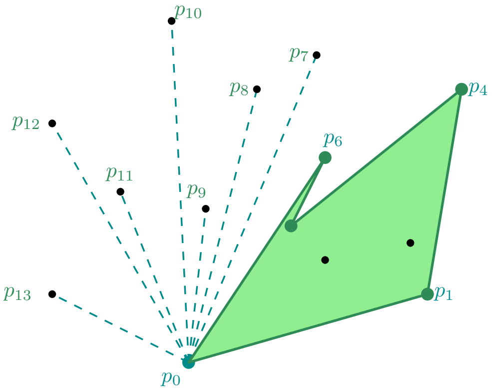
Enveloppe convexe de points

Exemple :



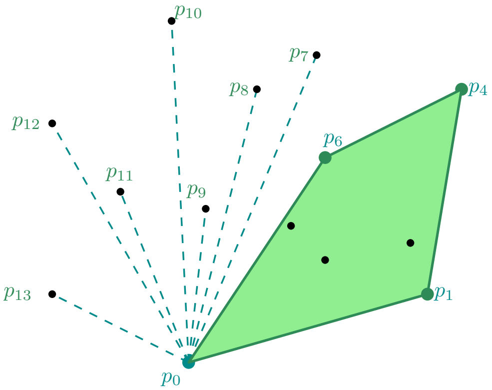
Enveloppe convexe de points

Exemple :



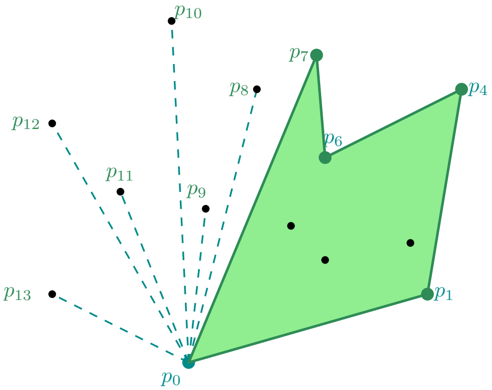
Enveloppe convexe de points

Exemple :



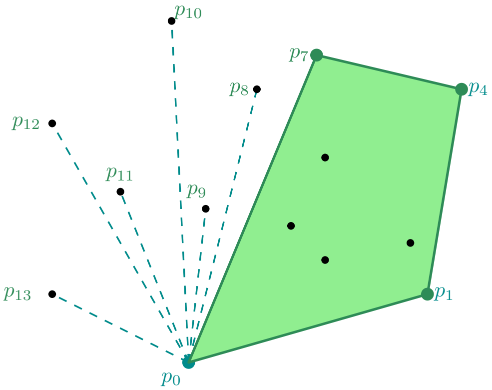
Enveloppe convexe de points

Exemple :

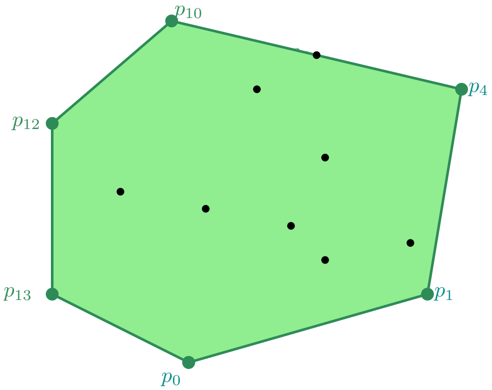


Enveloppe convexe de points

Exemple :



Exemple :



Théorème

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

Théorème

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

Démonstration

Théorème

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

Démonstration

- point le plus bas : $\Theta(n)$

Théorème

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

Démonstration

- point le plus bas : $\Theta(n)$
- tri selon l'angle : $\Theta(n \log n)$

Enveloppe convexe de points

Théorème

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

Démonstration

- point le plus bas : $\Theta(n)$
- 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

Applications du tri en géométrie : 2. Points les plus proches

distance_minimale(L)

étant donné une liste L de points du plan, déterminer la distance minimale entre deux éléments de L

distance_minimale(L)

étant donné une liste L de points du plan, déterminer la distance minimale entre deux éléments de L

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

distance_minimale(L)

étant donné une liste L de points du plan, déterminer la distance minimale entre deux éléments de L

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

Lemme

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

distance_minimale(L)

étant donné une liste L de points du plan, déterminer la distance minimale entre deux éléments de L

Approche diviser pour régner :

distance_minimale(L)

étant donné une liste `L` de points du plan, déterminer la distance minimale entre deux éléments de `L`

Approche diviser pour régner :

- séparer `L` en deux sous-listes `gauche` et `droite`

distance_minimale(L)

étant donné une liste `L` de points du plan, déterminer la distance minimale entre deux éléments de `L`

Approche diviser pour régner :

- séparer `L` en deux sous-listes `gauche` et `droite`
- calculer `d1 = distance_minimale(gauche)`
  et `d2 = distance_minimale(droite)`

Applications du tri en géométrie : 2. Points les plus proches

distance_minimale(L)

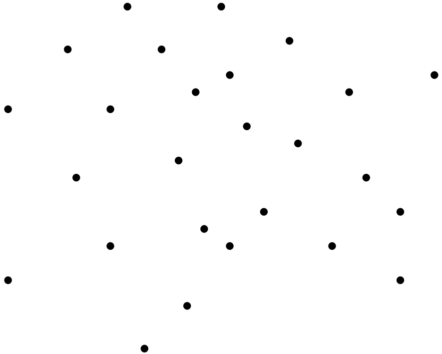
étant donné une liste `L` de points du plan, déterminer la distance minimale entre deux éléments de `L`

Approche diviser pour régner :

- séparer `L` 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)`

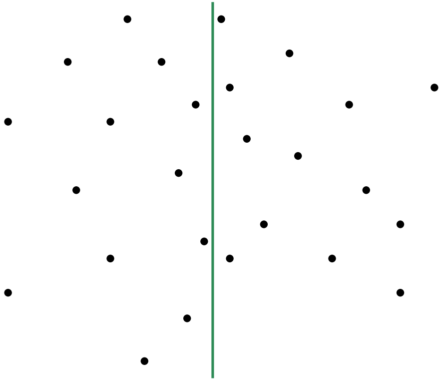
Points les plus proches

Exemple



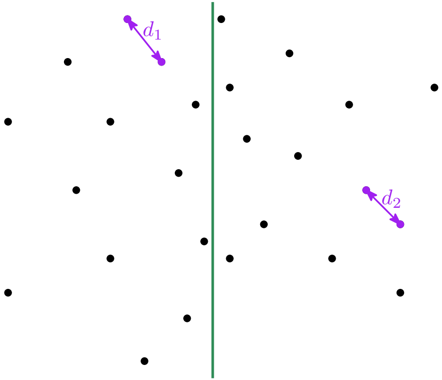
Points les plus proches

Exemple



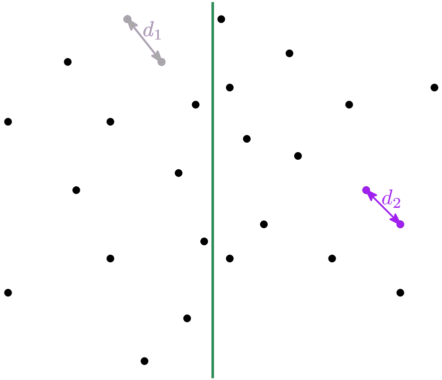
Points les plus proches

Exemple



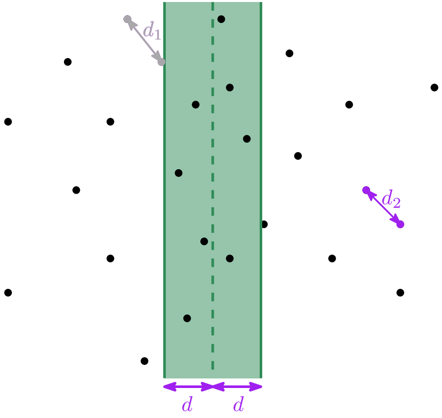
Points les plus proches

Exemple

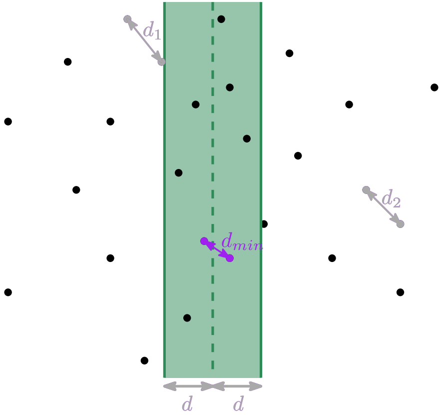


Points les plus proches

Exemple

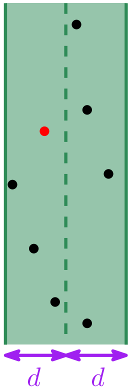


Exemple

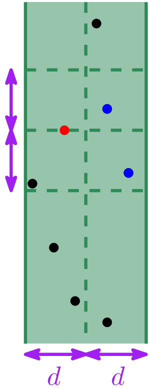


Points les plus proches

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

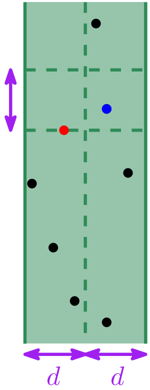


Comment trouver (`p1`, `p2`) efficacement ?



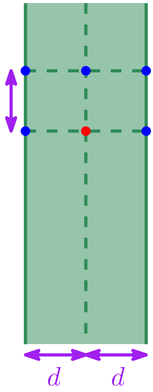
Points les plus proches

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



Points les plus proches

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

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

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

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

Applications du tri n° 3 : Recherche de motif dans un texte

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

Applications du tri n° 3 : Recherche de motif dans un texte

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

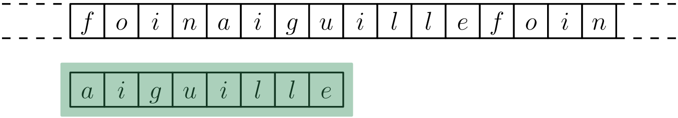
botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin botte de foin aiguille botte de foin ...

Applications du tri n° 3 : Recherche de motif dans un texte

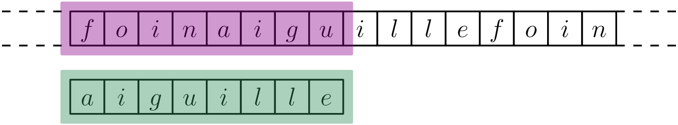
Étant donné un (petit) `motif` et un (corpus de) (long(s)) `texte`(s), déterminer si `motif` apparaît dans `texte`

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 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 ...

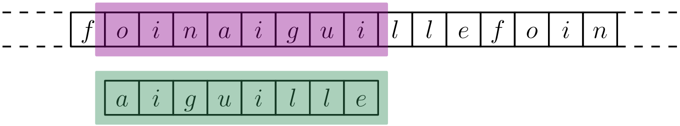
Algorithme naïf



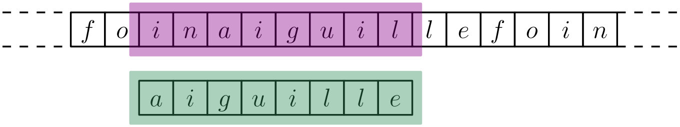
Algorithme naïf



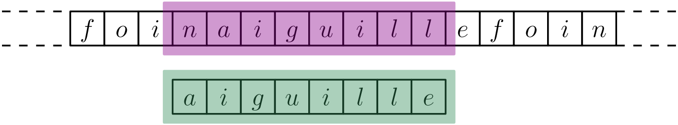
Algorithme naïf



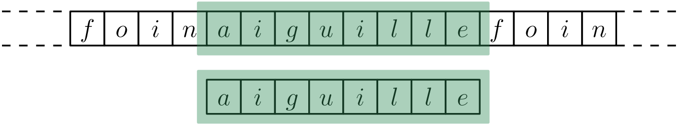
Algorithme naïf

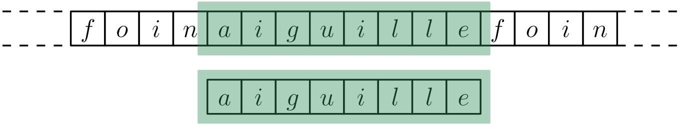


Algorithme naïf



Algorithme naïf





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

peut-on faire mieux ?

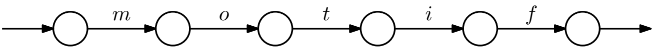
Recherche de motif dans un texte

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 ?

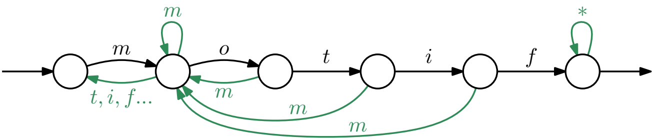
Cas de recherches répétées d'un motif

Réponse standard : avec un automate



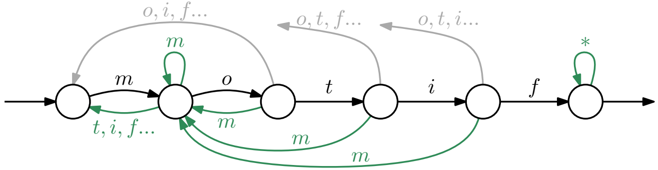
Cas de recherches répétées d'un motif

Réponse standard : avec un automate



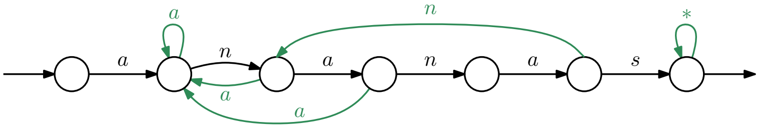
Cas de recherches répétées d'un motif

Réponse standard : avec un automate



Cas de recherches répétées d'un motif

Réponse standard : avec un automate

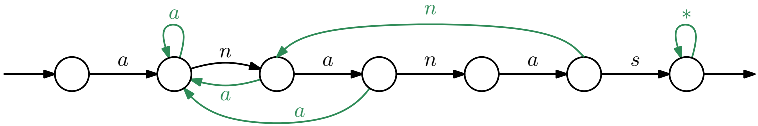


avantages :

- 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 : si l'alphabet a k lettres,

- (coût de la construction de l'automate) $\Theta(mk)$
- complexité en espace : nécessité de stocker un automate de taille $\Theta(mk)$

Indexation de textes

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

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

Indexation de textes

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

Cahier des charges : moyennant un prétraitement éventuel de `texte` (*indexation*), pouvoir faire des recherches 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 des recherches 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]
```

Exploitation de la table des suffixes

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

Exploitation de la table des suffixes

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

Coût de la comparaison de deux mots de longueurs ℓ_1 et ℓ_2 :

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

Exploitation de la table des suffixes

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

Coût de la comparaison de deux mots de longueurs ℓ_1 et ℓ_2 :
$\Theta(\min(\ell_1, \ell_2))$

Coût du test `est_occurrence(i, motif, texte)` : $\Theta(m)$

Exploitation de la table des suffixes

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

Coût de la comparaison de deux mots de longueurs ℓ_1 et ℓ_2 :

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

Coût du test `est_occurrence(i, motif, texte)` : $\Theta(m)$

Coût de la recherche d'un motif : recherche dichotomique

$$\Longrightarrow \Theta(\log n) \text{ comparaisons}$$

$$\Longrightarrow \Theta(m \log n) \text{ (au pire)}$$

Exploitation de la table des suffixes

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

Coût de la comparaison de deux mots de longueurs ℓ_1 et ℓ_2 :

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

Coût du test `est_occurrence(i, motif, texte)` : $\Theta(m)$

Coût de la recherche d'un motif : recherche dichotomique

$$\Longrightarrow \Theta(\log n) \text{ comparaisons}$$

$$\Longrightarrow \Theta(m \log n) \text{ (au pire)}$$

Coût de la recherche de toutes les occurrences d'un motif :

$$\Theta(m \log n)$$

Construction de la table des suffixes

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

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 $\Theta(n)$

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

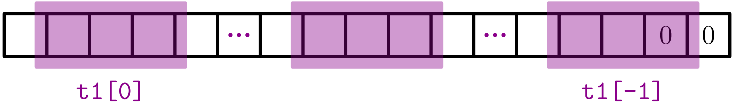
1. ajouter encore 2 caractères `'0'` à `texte`
2. définir `t1` et `t2` : mots sur l'alphabet A^3 dont la « projection » sur A^* est respectivement `texte[1:-1]` et `texte[2:]` :

															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

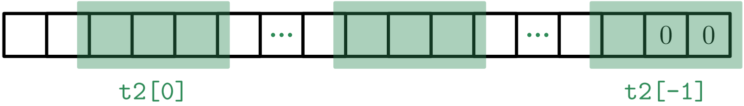
1. ajouter encore 2 caractères `'0'` à `texte`
2. définir `t1` et `t2` : mots sur l'alphabet A^3 dont la « projection » sur A^* est respectivement `texte[1:-1]` et `texte[2:]` :



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 :

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