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 2021-2022

Résumé des épisodes précédents

Qualités et défauts des algorithmes de tri classiques

Tri par fusion

- $\Theta(n \log n)$ comparaisons *au pire* (mais *dans tous les cas*),
- la *constante cachée* dans le Θ est importante,
- *stable* mais *pas en place* : complexité en espace $\in \Theta(n)$

Tri par insertion

- $\Theta(n^2)$ comparaisons *au pire* et *en moyenne*,
- $\Theta(n)$ comparaisons *au mieux* (CNS : $O(n)$ inversions),
- *stable* et *en place*

Tri rapide

- $\Theta(n^2)$ comparaisons *au pire*
- $\Theta(n \log n)$ comparaisons *en moyenne*... et *au mieux*,
- version naïve : *stable* mais *pas en place*, mauvais cas « assez probables »,
- version en place et randomisée : *en place* mais *pas stable*, mauvais cas sans caractéristiques particulières (donc peu probables)

Comment conjuguer ces qualités ?

Tri par comparaisons « idéal » :

- pire cas (et cas moyen) en $\Theta(n \log n)$,
- meilleur cas en $\Theta(n)$ (correspondant à des cas « probables en pratique »),
- en place,
- stable.

pour s'en approcher, on peut concevoir des tris hybrides : tris utilisant des mécanismes inspirés de plusieurs algorithmes de tri différents

- SedgeSort (hybride de tri rapide et de tri par insertion),
- TimSort (hybride de tri fusion et de tri par insertion),
- IntroSort (hybride de tri rapide et de tri par tas)...

et il est parfois possible de tirer parti des caractéristiques des données à trier pour sortir du cadre des tris par comparaisons : tri par dénombrement, tri par paquets, tri par base (RadixSort)...

Tri par base (*radix sort*)

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

```
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 $\Theta(nk)$

Complément : la sélection rapide

Rang

l'élément de `rang k` d'un tableau `T` est l'unique `x` de `T` tel que

- `T` contient au plus `k` − 1 éléments strictement plus petits que `x`
- `T` contient au plus `len(T)` − `k` éléments strictement plus grands que `x`

Rang

si `T` est un tableau ***sans doublon***, l'élément de `rang k` de `T` est l'unique `x` de `T` tel que

- `T` contient `k` − 1 éléments plus petits que `x`
- `T` contient `len(T)` − `k` éléments plus grands que `x`

Rang

si `T` est un tableau ***sans doublon***, l'élément de `rang k` de `T` est l'unique `x` de `T` tel que

- `T` contient `k` − 1 éléments plus petits que `x`
- `T` contient `len(T)` − `k` éléments plus grands que `x`

Cas particuliers

- ***si*** `T` est trié : `T[k-1]`
- élément de rang 1 : `minimum(T)`
- élément de rang `len(T)` : `maximum(T)`
- élément « du milieu » : `médian(T)` (ou `médiane(T)`)
  si $n =$ `len(T)` impair : rang $\frac{1}{2}(n+1)$
  (si n pair : rang $\frac{1}{2}n$ ou $\frac{1}{2}n+1$)

`selection(T, k)`

étant donné un tableau `T` et un entier `k`, déterminer l'élément de rang `k` de `T`

`selection(T, k)`

étant donné un tableau `T` et un entier `k`, déterminer l'élément de rang `k` de `T`

Solution n° 1

- trier `T`
- retourner `T[k-1]`

`selection(T, k)`

étant donné un tableau `T` et un entier `k`, déterminer l'élément de rang `k` de `T`

Solution n° 1

- trier `T`
- retourner `T[k-1]`

$\Longrightarrow \Theta(n \log n)$ *comparaisons (au pire)*

minimum(T)

étant donné un tableau T, déterminer le plus petit élément de T

```
def min(T) :
  tmp = T[0]
  for elt in T :
    if elt < tmp : tmp = elt
  return tmp
```

$\Longrightarrow$ $n-1$ *comparaisons (exactement)*

maximum(T)

étant donné un tableau T, déterminer le plus grand élément de T

```
def max(T) :
  tmp = T[0]
  for elt in T :
    if elt > tmp : tmp = elt
  return tmp
```

$\Longrightarrow$ $n-1$ *comparaisons (exactement)*

Sélection – cas général

```
def selection(T, k) : # comme un tri par sélection interrompu
  for i in range(k) :
    tmp = i
    for j in range(i, len(T)) :
      if T[j] < T[tmp] : tmp = j
    T[i], T[tmp] = T[tmp], T[i]
  return T[k-1]
```

Sélection – cas général

```
def selection(T, k) : # comme un tri par sélection interrompu
  for i in range(k) :
    tmp = i
    for j in range(i, len(T)) :
      if T[j] < T[tmp] : tmp = j
    T[i], T[tmp] = T[tmp], T[i]
  return T[k-1]
```

$\Longrightarrow$ kn *comparaisons (environ)*

- si `k` est petit, c'est sensiblement mieux que $\Theta(n \log n)$
- si `k` est en $\Theta(n)$, c'est sensiblement moins bien

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si `r` $=$ `k` : le pivot est l'élément de rang `k` $\Longrightarrow$ recherche terminée

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si `r` $=$ `k` : le pivot est l'élément de rang `k` $\Longrightarrow$ recherche terminée
- si `r` $>$ `k` : le pivot est supérieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche à gauche

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si `r` $=$ `k` : le pivot est l'élément de rang `k` $\Longrightarrow$ recherche terminée
- si `r` $>$ `k` : le pivot est supérieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche à gauche
- si `r` $<$ `k` : le pivot est inférieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche ***à droite***

Sélection rapide (*Quickselect*)

Idée : utiliser le partitionnement du tri rapide

Que conclure de la position `r`(-1) du pivot retournée par `partition(T)` ?

- si `r` $=$ `k` : le pivot est l'élément de rang `k` $\Longrightarrow$ recherche terminée
- si `r` $>$ `k` : le pivot est supérieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche à gauche
- si `r` $<$ `k` : le pivot est inférieur à l'élément de rang `k`
  $\Longrightarrow$ poursuivre la recherche ***à droite***

$\Longrightarrow$ dans tous les cas, (au plus) un seul appel récursif est nécessaire

Sélection rapide (*Quickselect*)

```
def selection_rapide(T, k) :
  if len(T) == 1 : return T[0] if k == 1 else None

  # version naïve
  pivot, gauche, droite = partition(T)
  rang_pivot = len(gauche) + 1

  if rang_pivot == k :
    return pivot
  elif rang_pivot > k :
    return selection_rapide(gauche, k)
  else :
```

Sélection rapide (*Quickselect*)

```
def selection_rapide(T, k) :
  if len(T) == 1 : return T[0] if k == 1 else None

  # version naïve
  pivot, gauche, droite = partition(T)
  rang_pivot = len(gauche) + 1

  if rang_pivot == k :
    return pivot
  elif rang_pivot > k :
    return selection_rapide(gauche, k)
  else :
    return selection_rapide(droite, k - rang_pivot)
```

Sélection rapide (*Quickselect*)

```
def selection_rapide_en_place(T, k, deb=0, fin=None) :
  if fin is None : fin = len(T)
  if fin-deb == 1 : return T[0] if k == 1 else None

  indice_pivot = partition_en_place(T, debut, fin)
  rang_pivot = indice_pivot + 1

  if rang_pivot == k :
    return T[indice_pivot]
  elif rang_pivot > k :
    return selection_rapide(T, k, deb, indice_pivot)
  else :
    return selection_rapide(T, k - rang_pivot, rang_pivot, fin)
```

Sélection rapide (*Quickselect*)

Complexité de `selection_rapide` au pire : $\Theta(n^2)$ comparaisons

Complexité de `selection_rapide` dans le meilleur des cas : $\Theta(n)$ comparaisons

Complexité de `selection_rapide` en moyenne (*admis*) : $\Theta(n)$ comparaisons

Sélection rapide (*Quickselect*)

Complexité de `selection_rapide` au pire : $\Theta(n^2)$ comparaisons

Complexité de `selection_rapide` dans le meilleur des cas : $\Theta(n)$ comparaisons

Complexité de `selection_rapide` en moyenne (*admis*) : $\Theta(n)$ comparaisons

En choisissant comme pivot la médiane des $\frac{n}{5}$ médianes de paquets de 5 éléments, on obtient un algorithme de complexité $\Theta(n)$ ***dans le pire des cas*** (admis)

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 $\mathcal{P}$

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

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

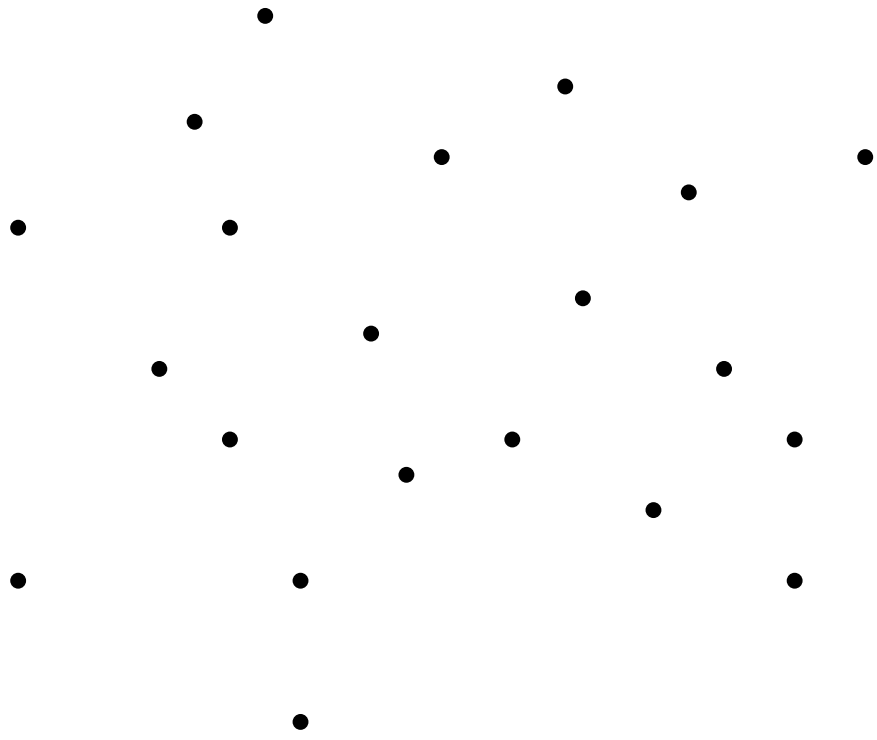
enveloppe convexe d'une partie $\mathcal{P}$ du plan : plus petite partie convexe $\mathcal{C}$ contenant $\mathcal{P}$

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)

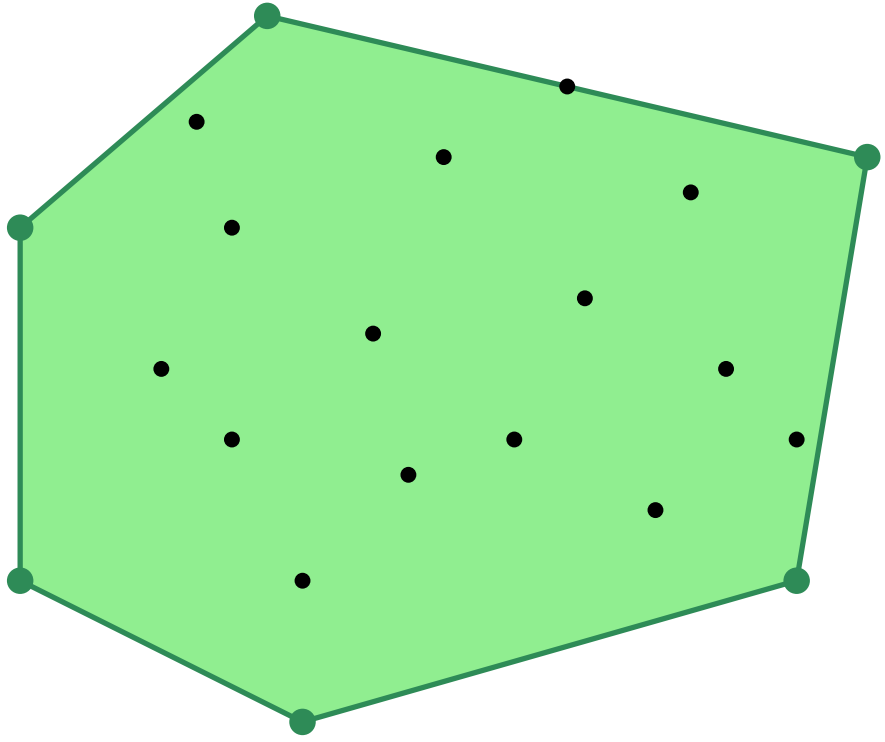
étant donné un `nuage` de points du plan, déterminer l'enveloppe convexe des points du `nuage`

Caractérisation des arêtes de l'enveloppe



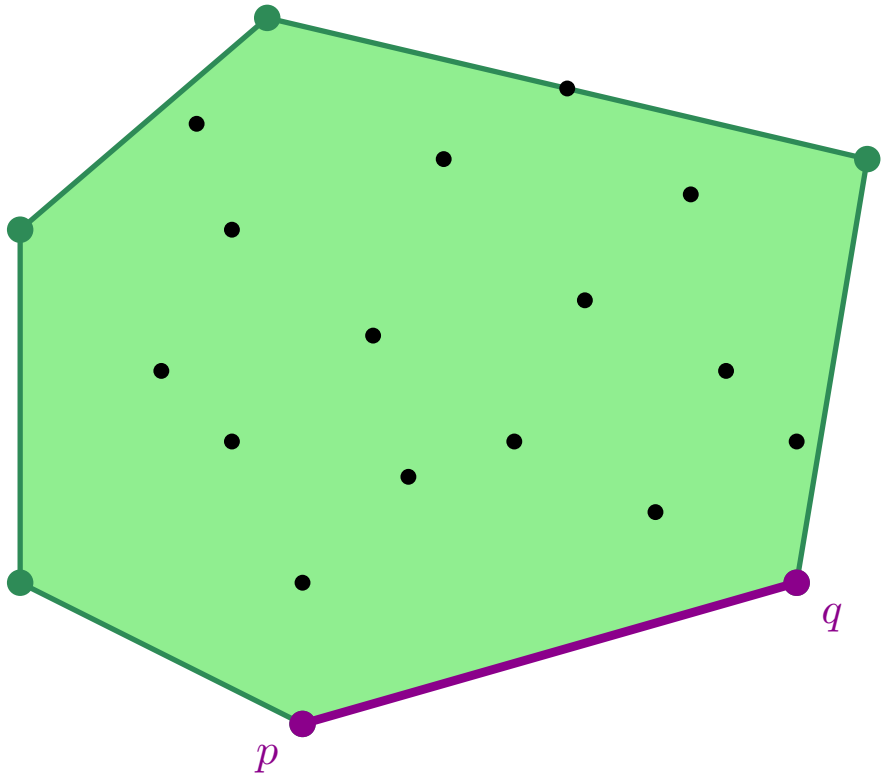
Un nuage de points

Caractérisation des arêtes de l'enveloppe



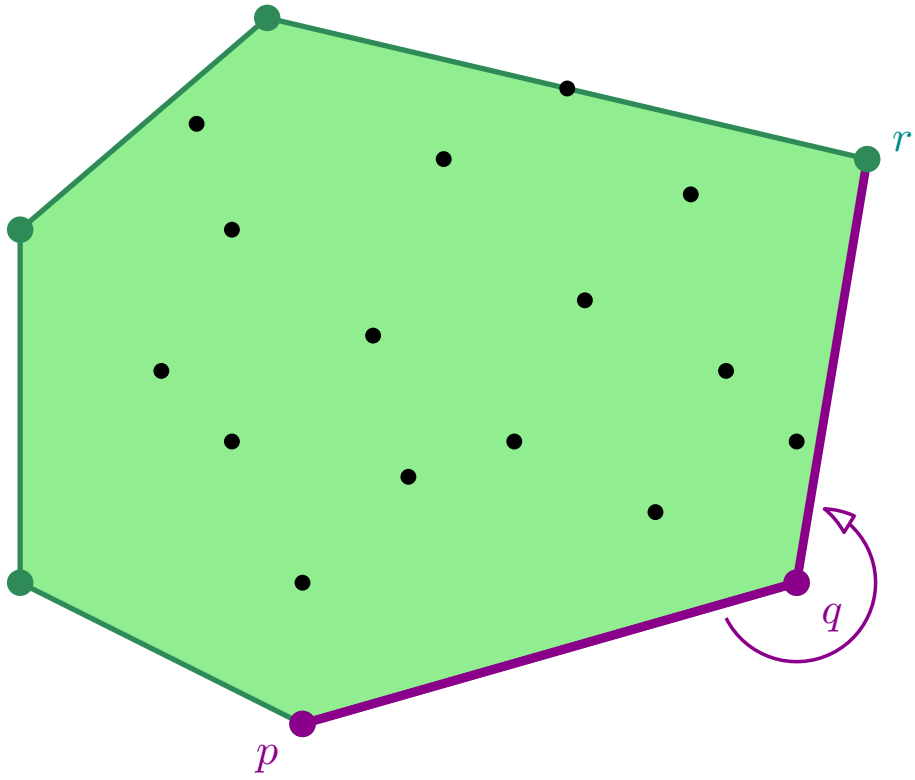
Son enveloppe convexe

Caractérisation des arêtes de l'enveloppe



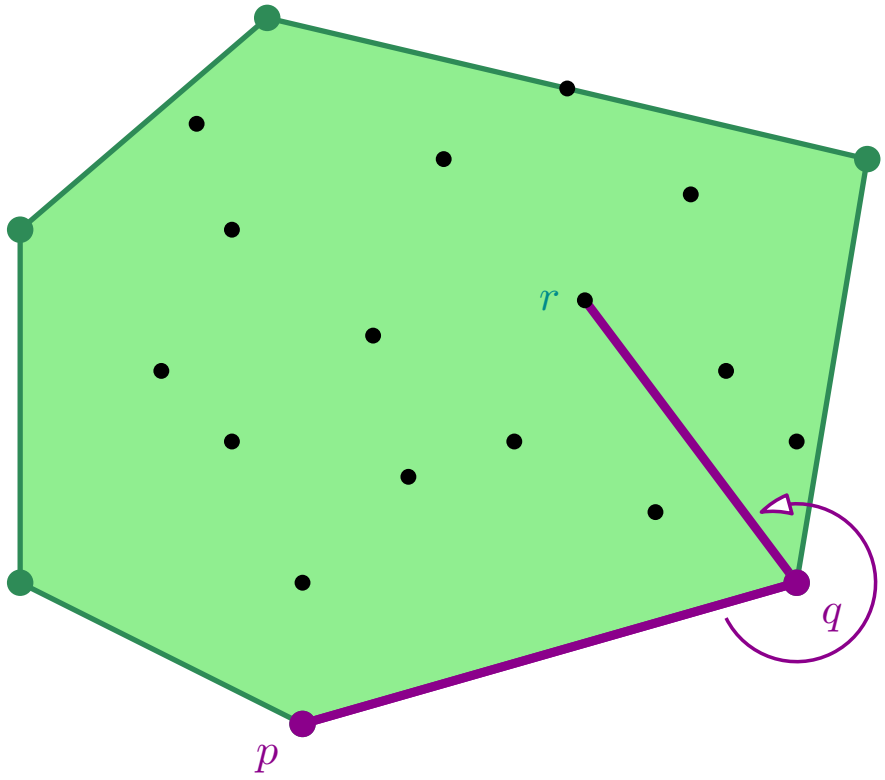
Une arête [pq] de l'enveloppe (dans le sens direct)

Caractérisation des arêtes de l'enveloppe



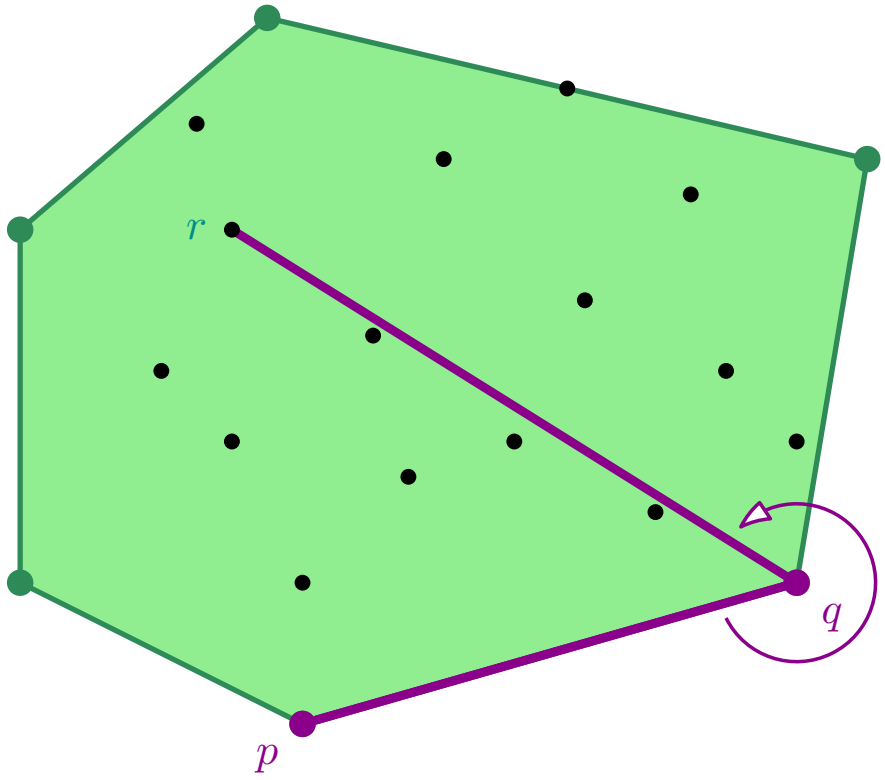
Tous les angles $\widehat{pqr}$ « tournent à gauche »

Caractérisation des arêtes de l'enveloppe



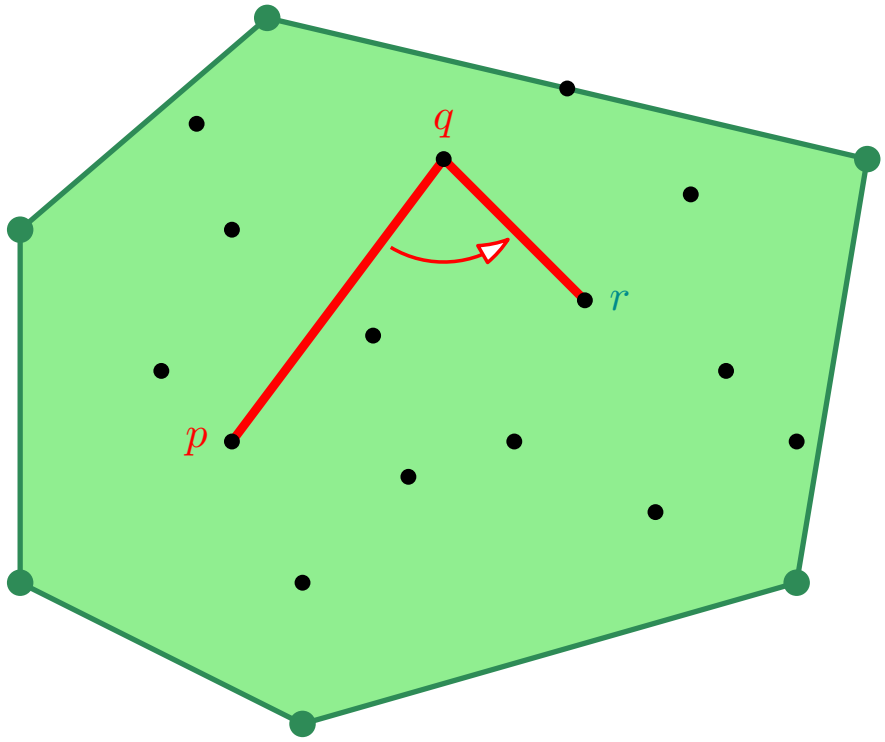
Tous les angles $\widehat{pqr}$ « tournent à gauche »

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)

Enveloppe convexe d'un nuage – par balayage

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

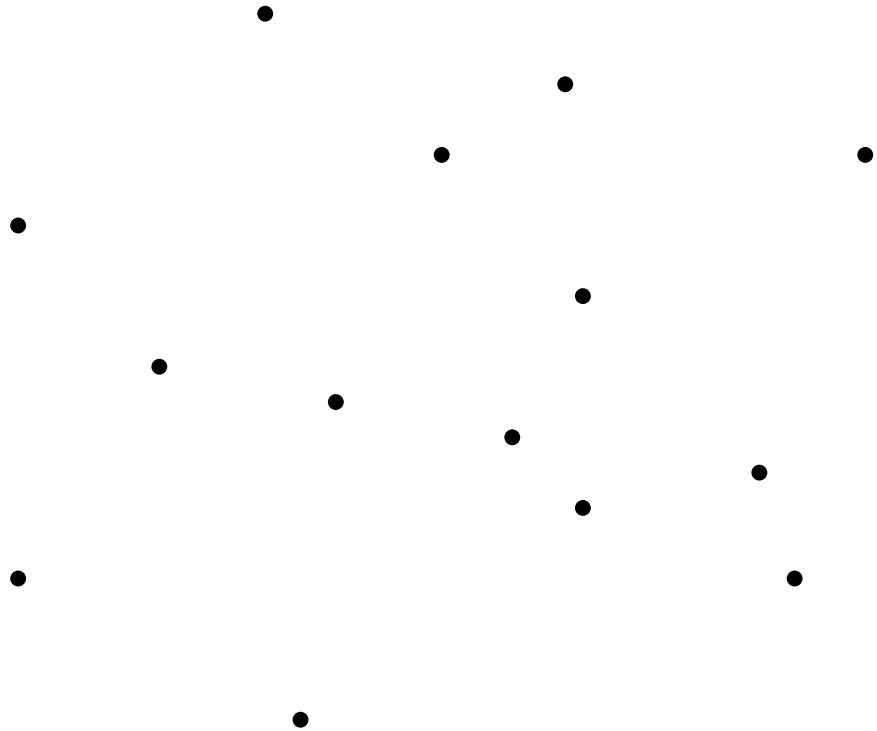
Plus précisément :

- 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 – $p_1, p_2, \ldots p_{n-1}$ 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 ?

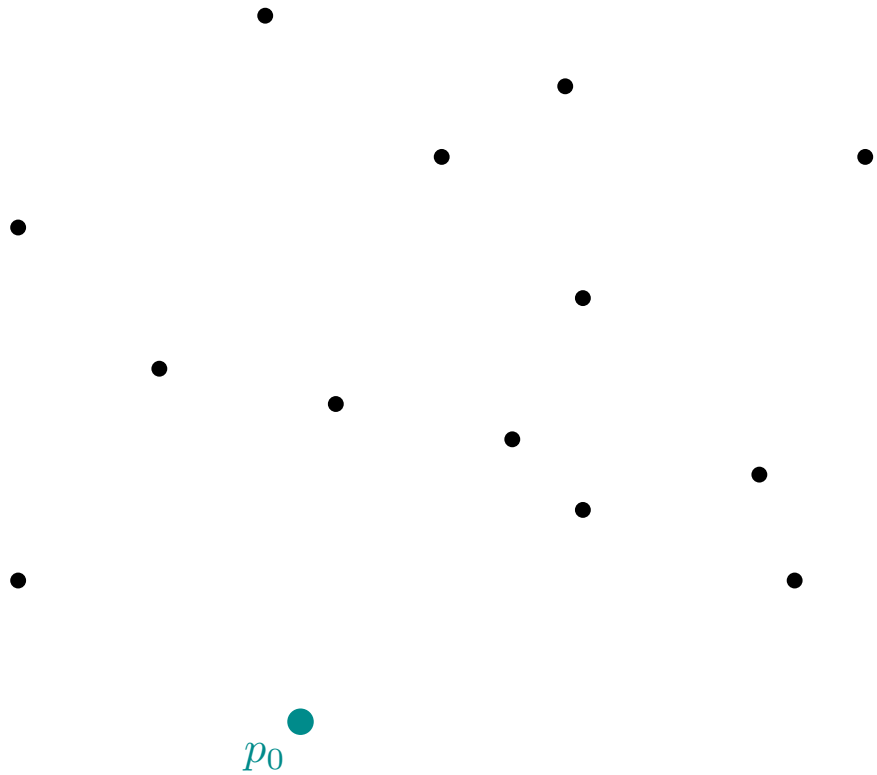
Enveloppe convexe d'un nuage – par balayage

Exemple :



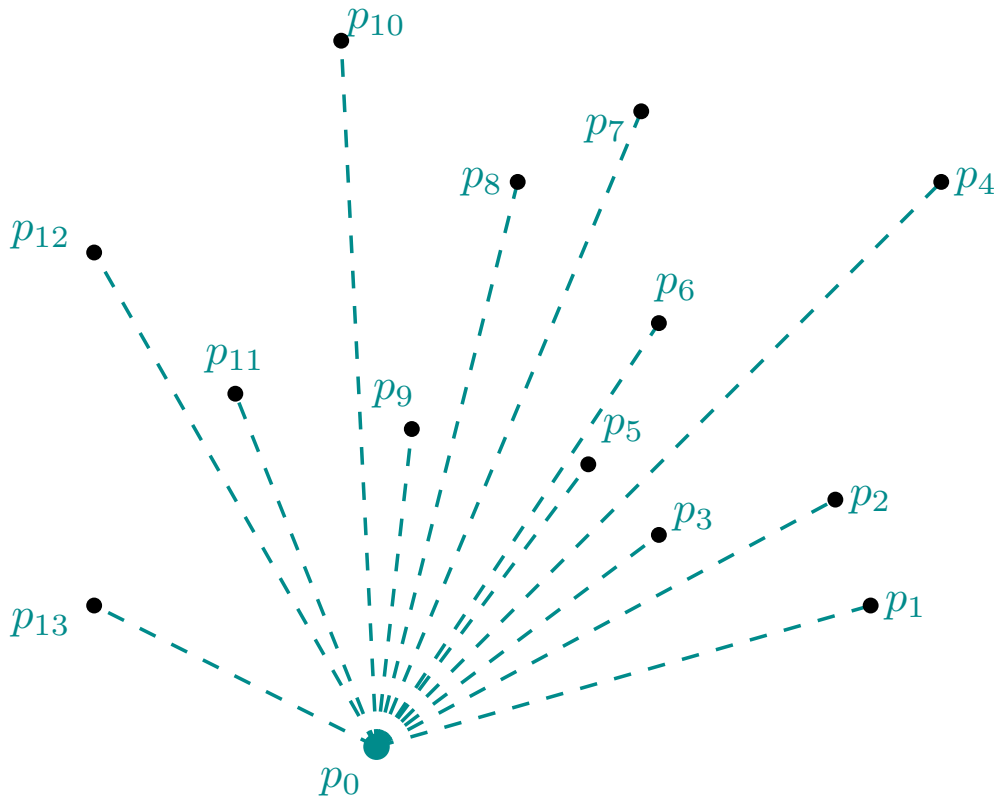
Enveloppe convexe d'un nuage – par balayage

Exemple :



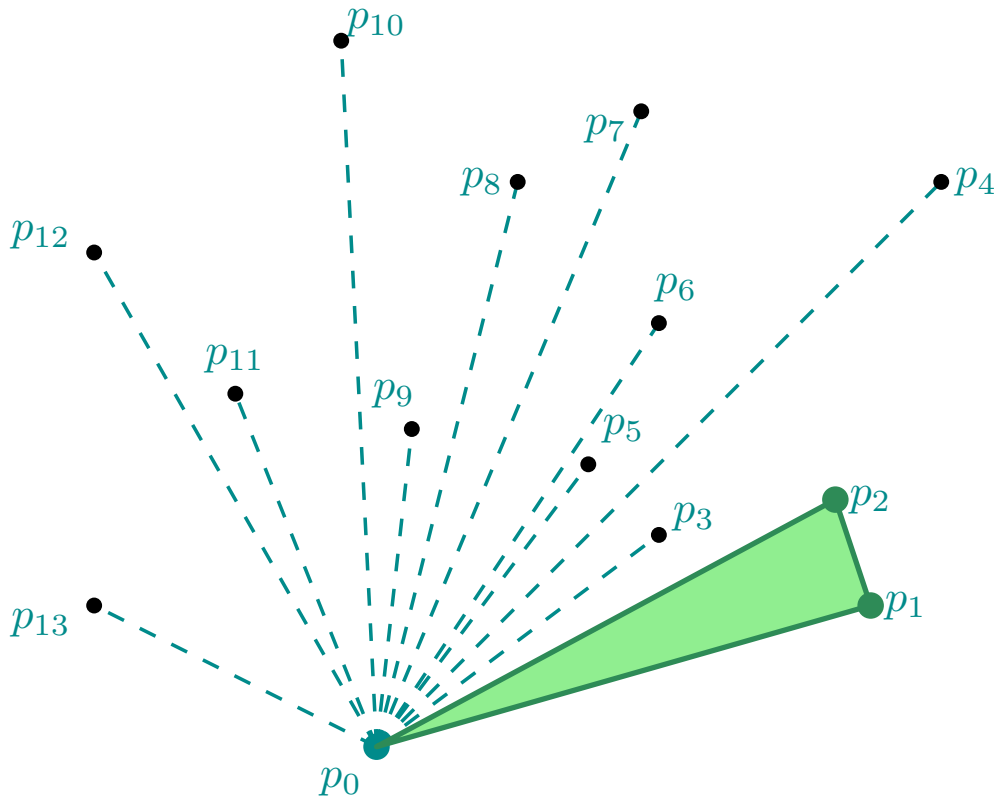
Enveloppe convexe d'un nuage – par balayage

Exemple :



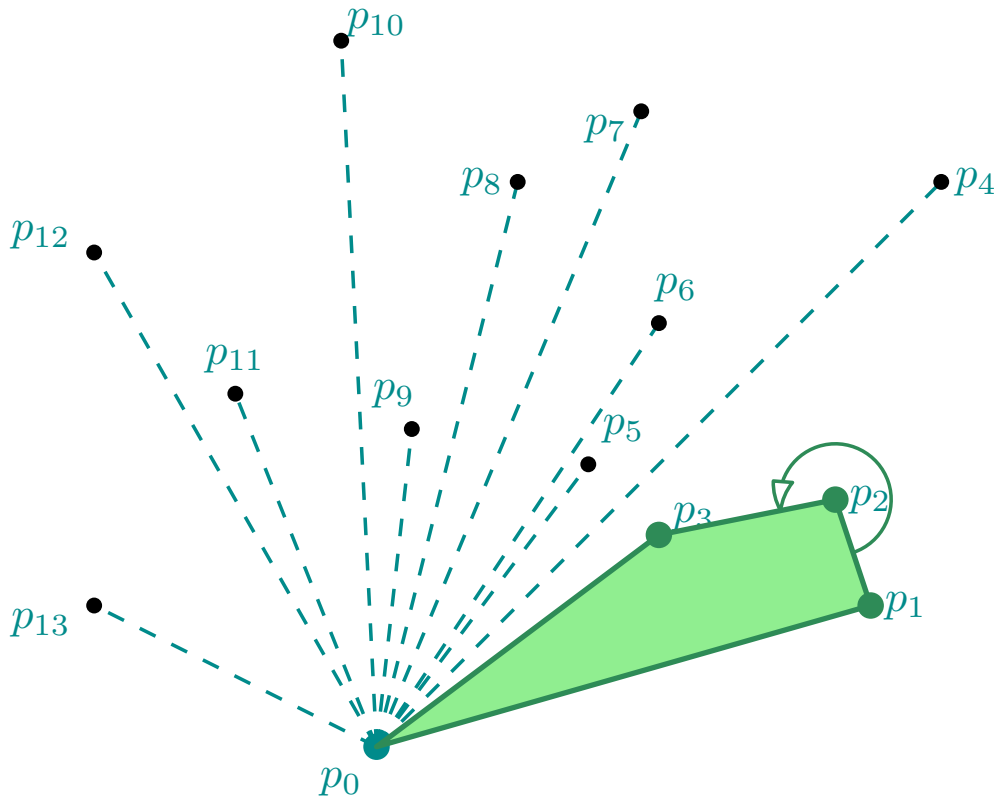
Enveloppe convexe d'un nuage – par balayage

Exemple :



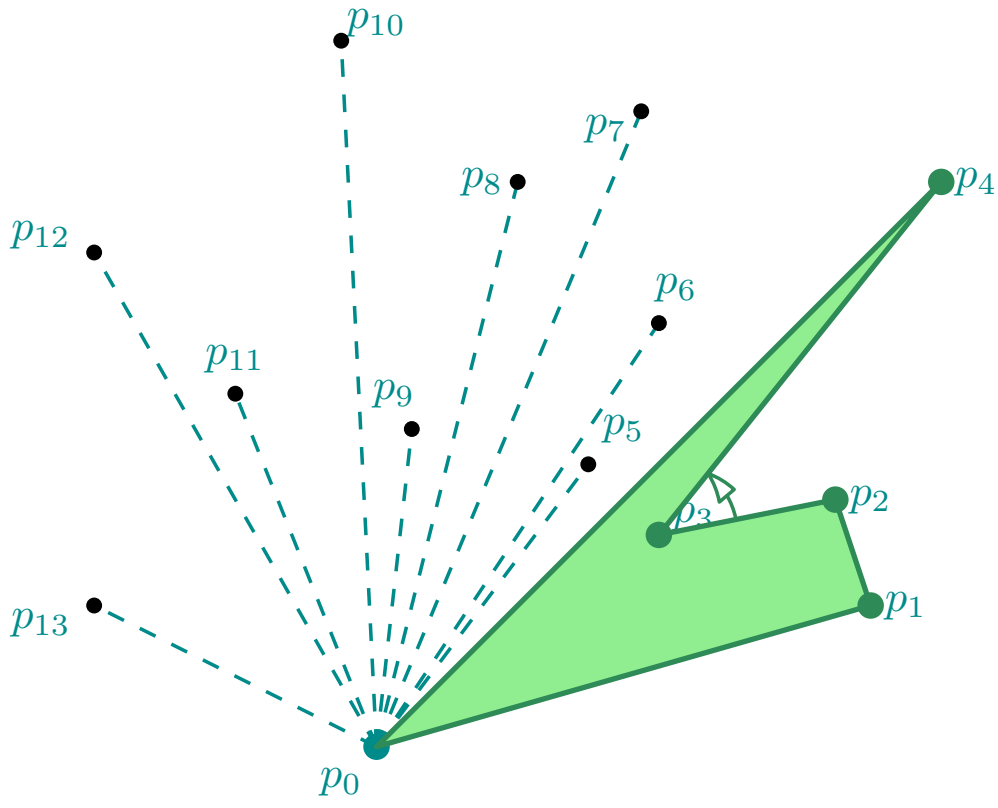
Enveloppe convexe d'un nuage – par balayage

Exemple :



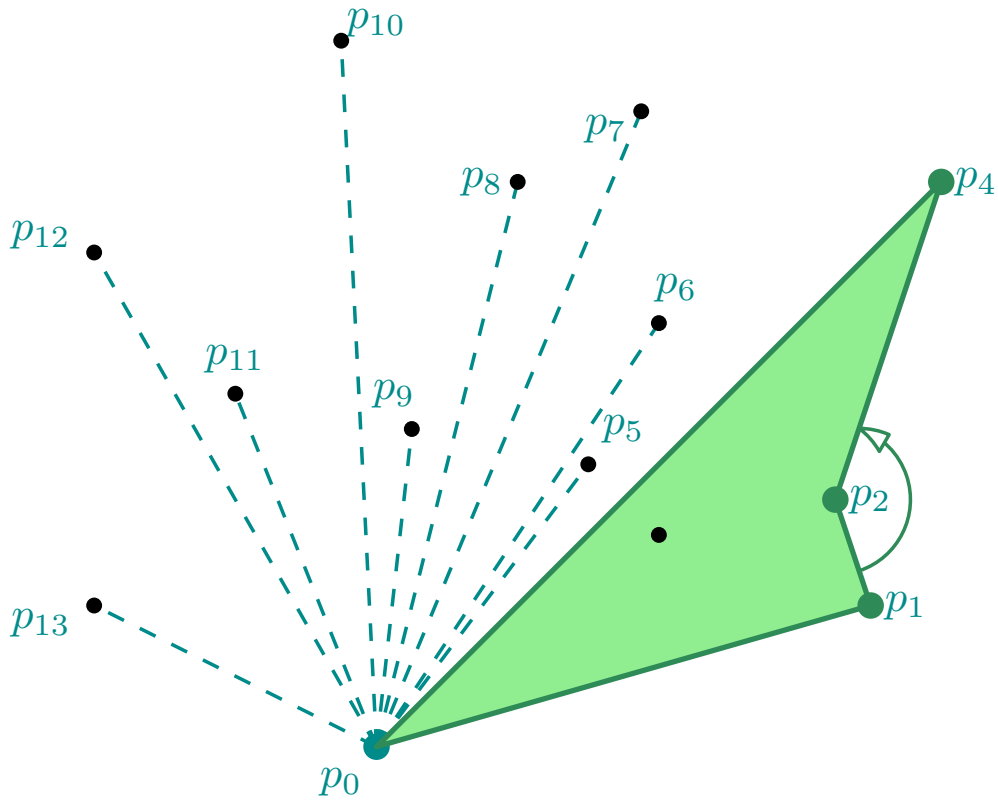
Enveloppe convexe d'un nuage – par balayage

Exemple :



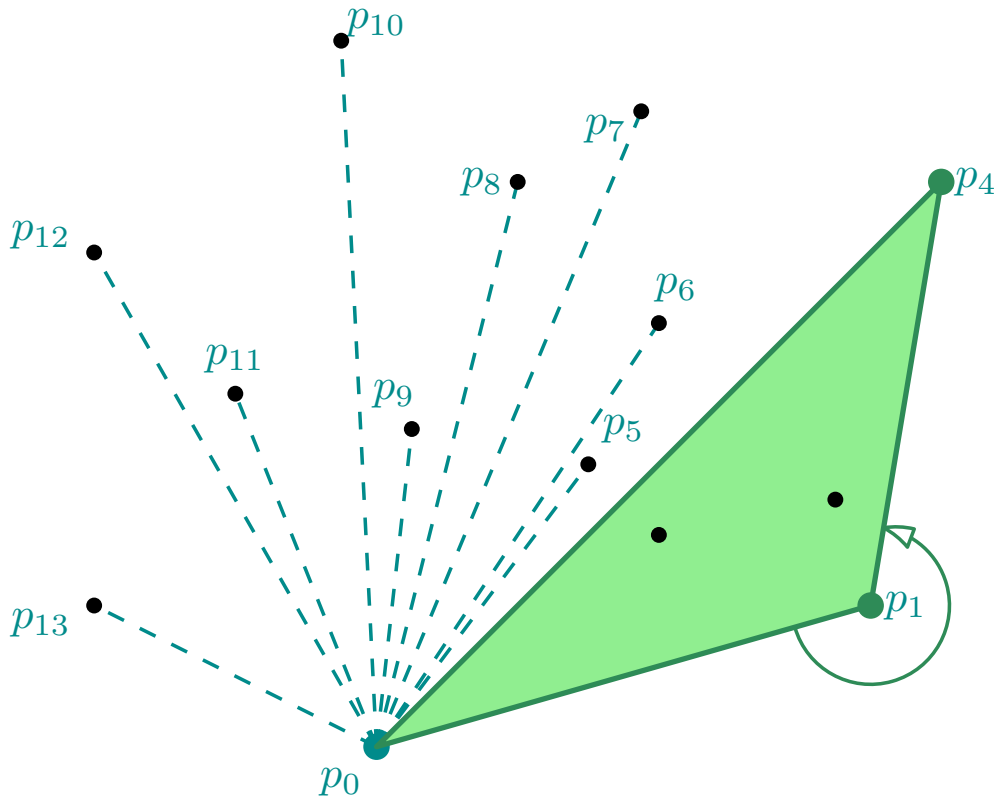
Enveloppe convexe d'un nuage – par balayage

Exemple :



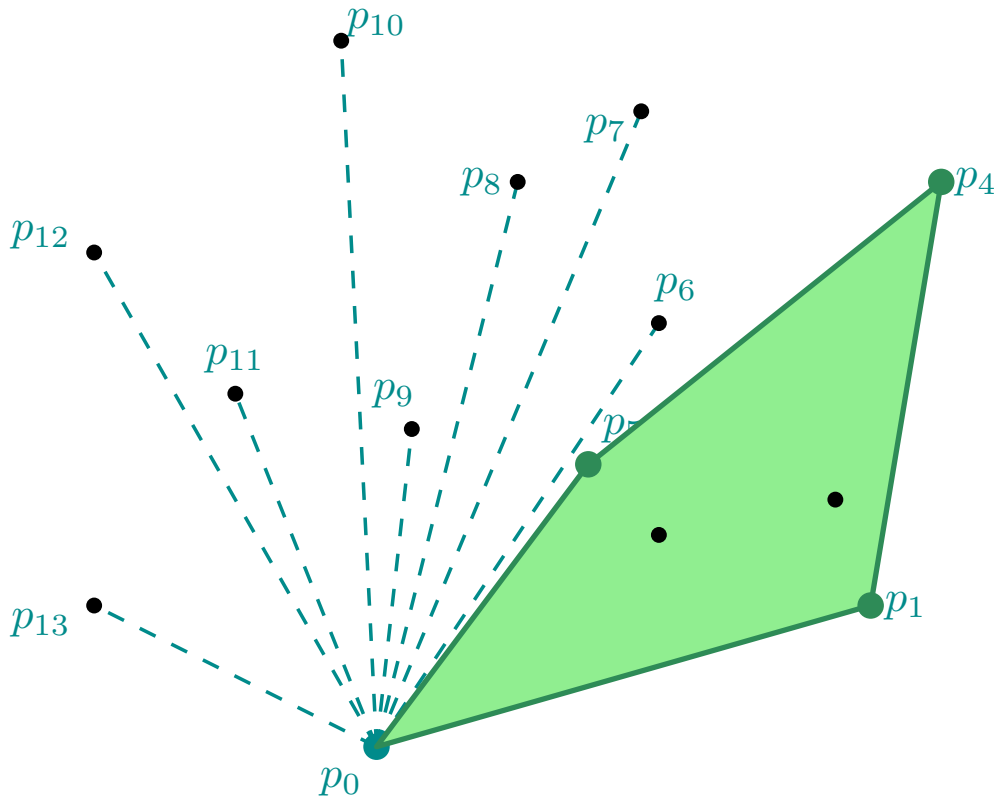
Enveloppe convexe d'un nuage – par balayage

Exemple :



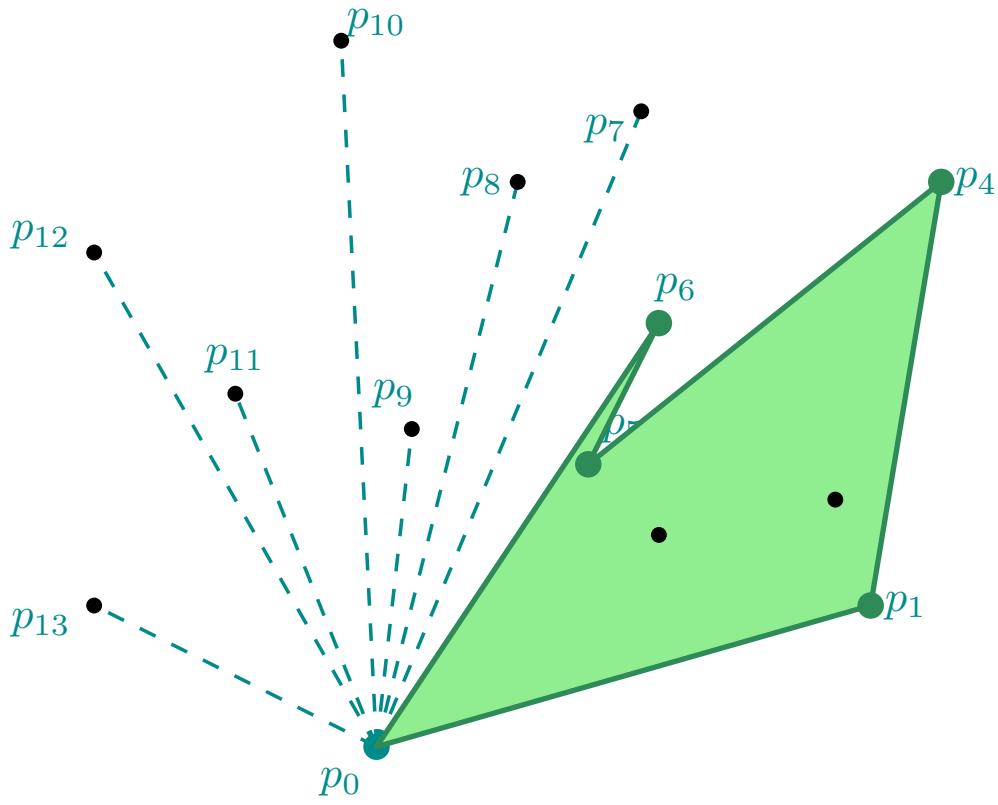
Enveloppe convexe d'un nuage – par balayage

Exemple :



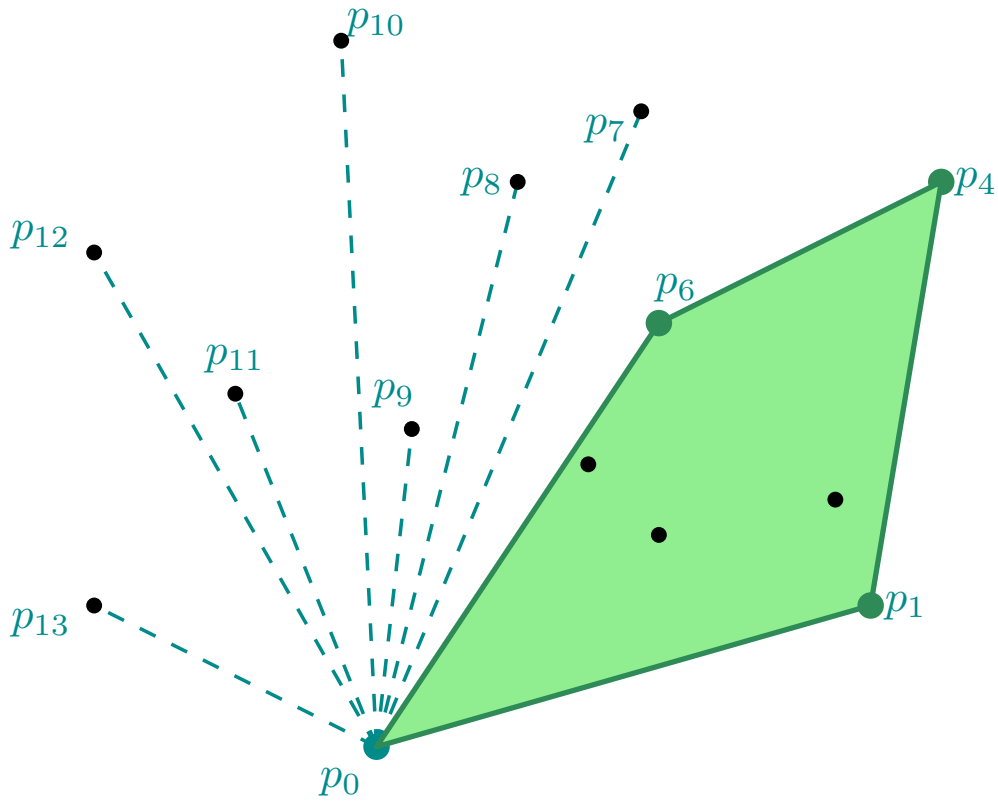
Enveloppe convexe d'un nuage – par balayage

Exemple :



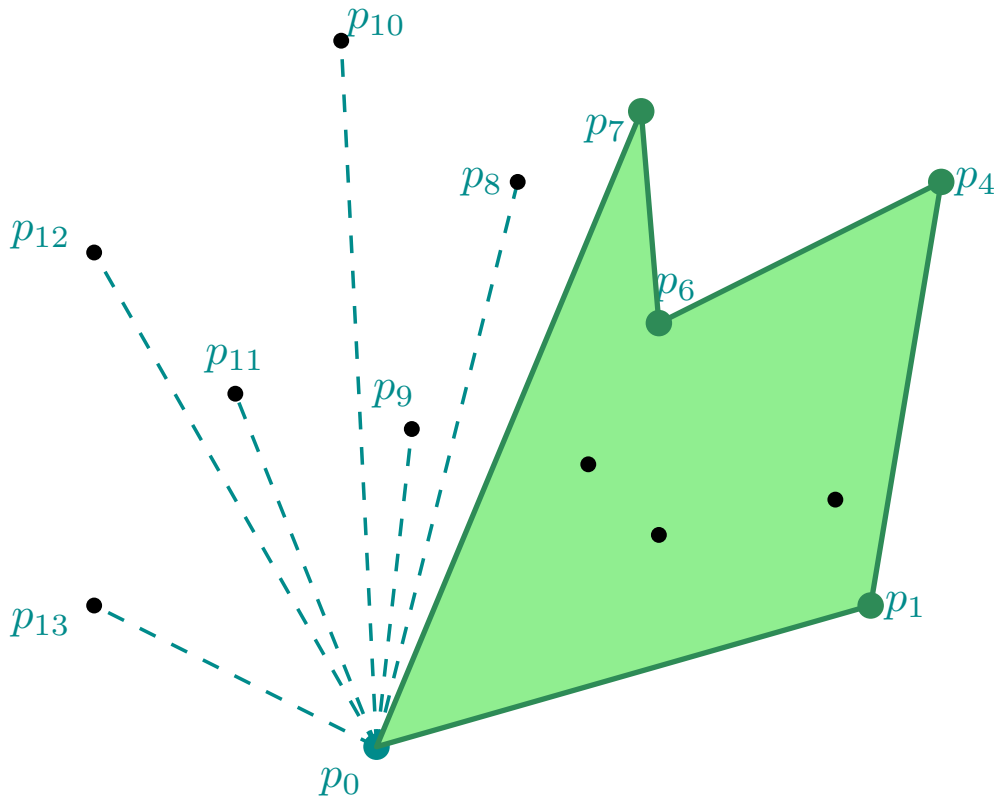
Enveloppe convexe d'un nuage – par balayage

Exemple :



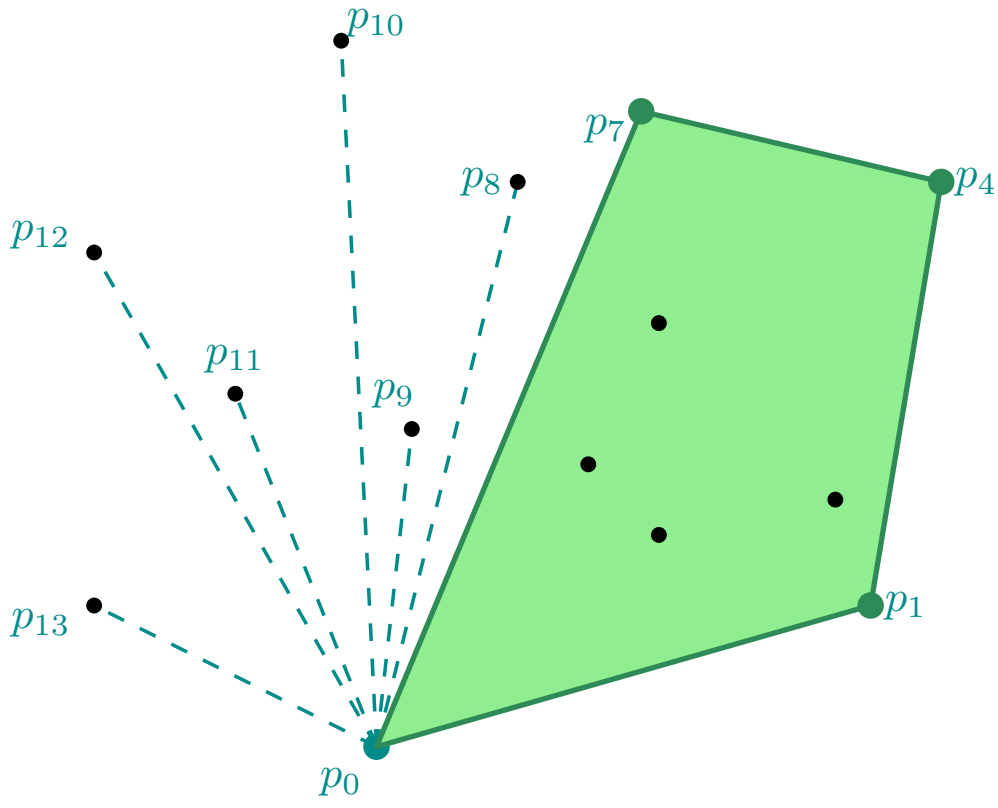
Enveloppe convexe d'un nuage – par balayage

Exemple :



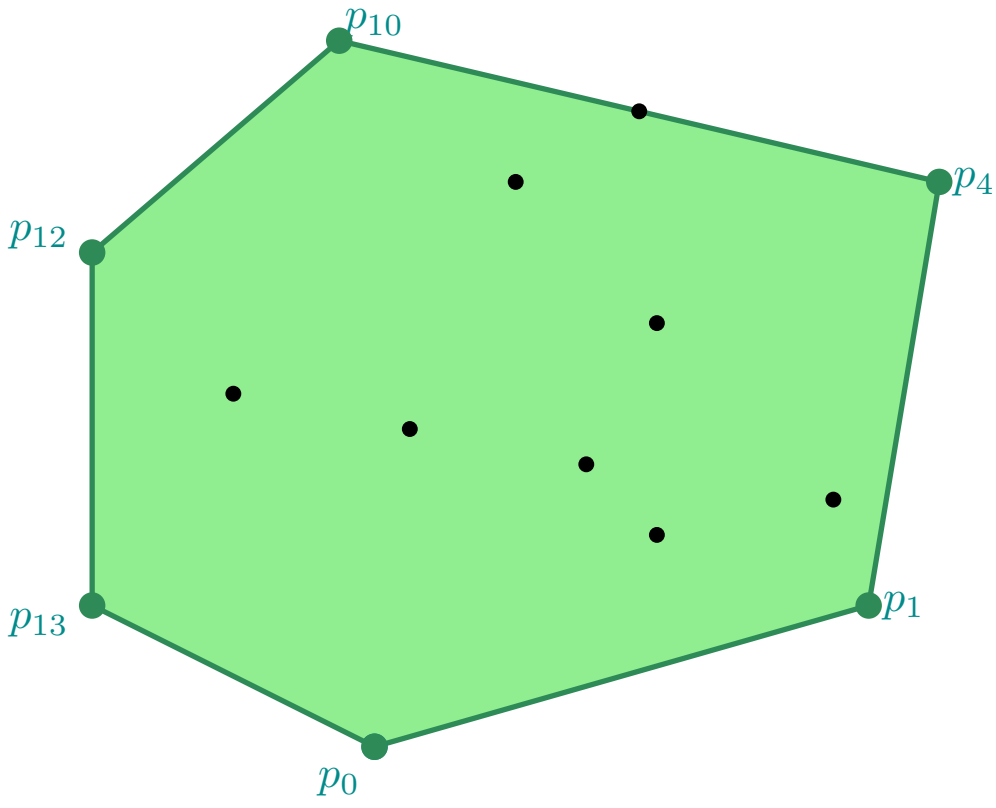
Enveloppe convexe d'un nuage – par balayage

Exemple :



Enveloppe convexe d'un nuage – par balayage

Exemple :



Enveloppe convexe d'un nuage – par balayage

```
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

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

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

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

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

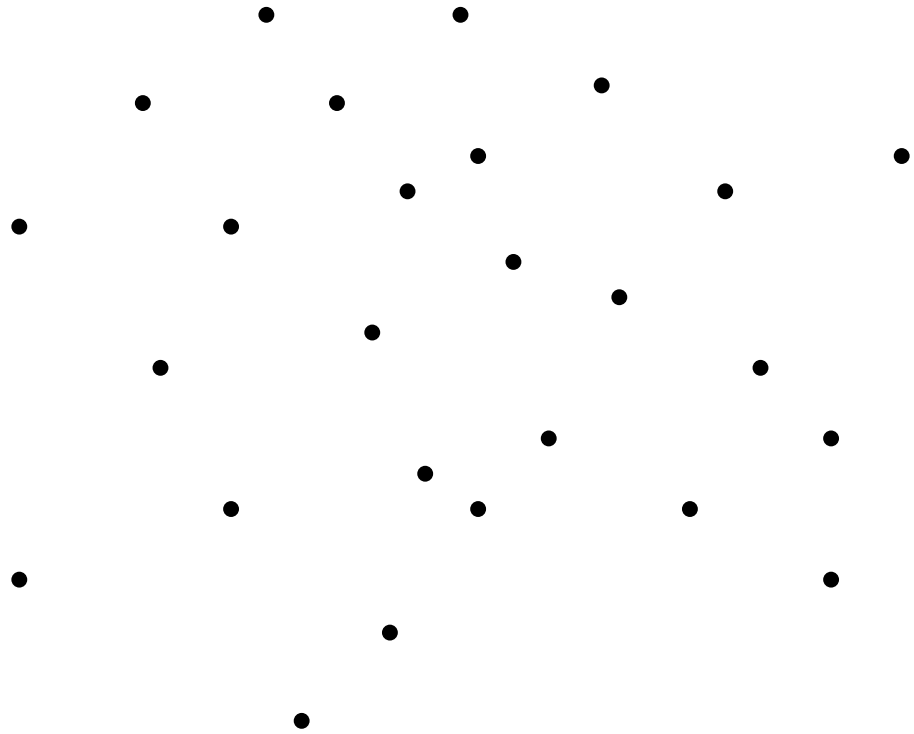
distance_minimale(nuage)

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

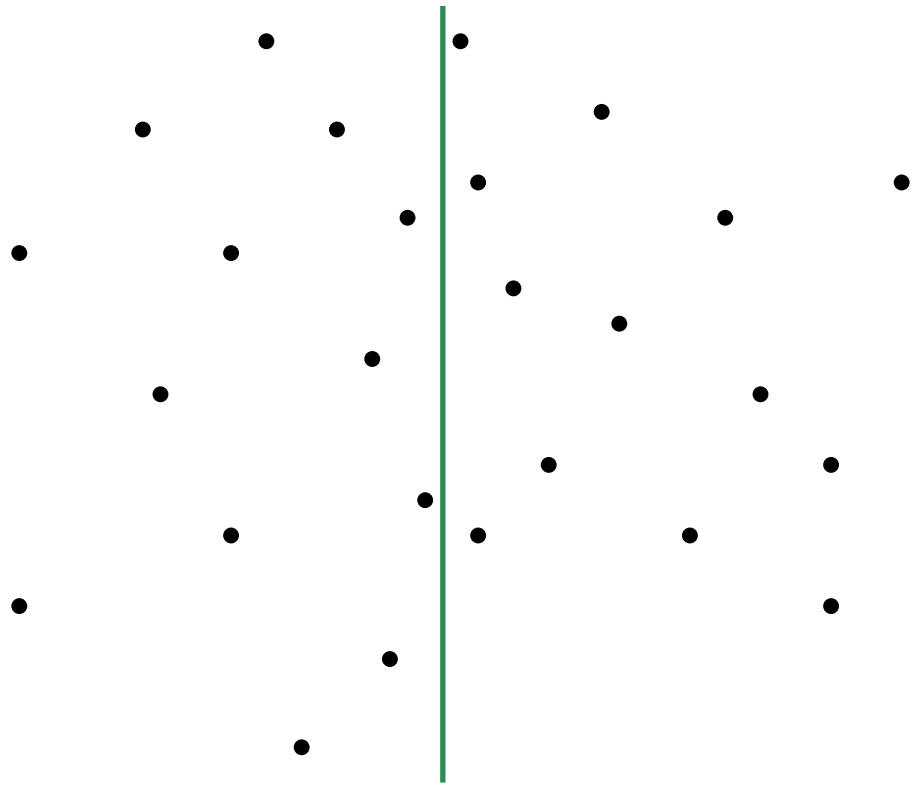
Approche « ***diviser pour régner*** » :

- 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 »*

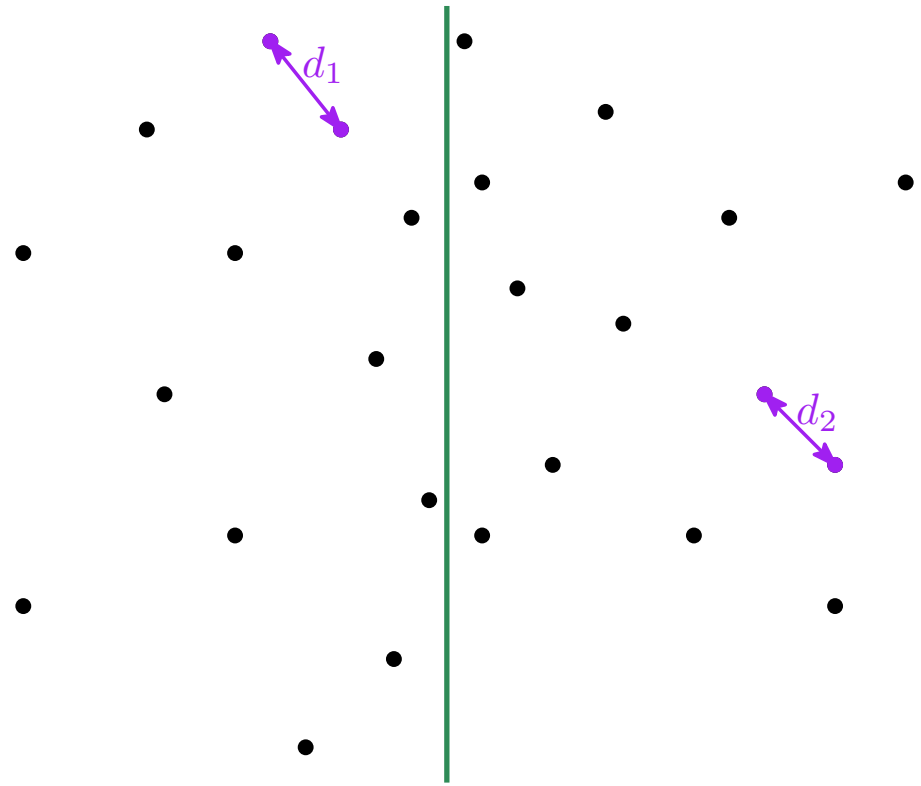


Points les plus proches – *« diviser pour régner »*



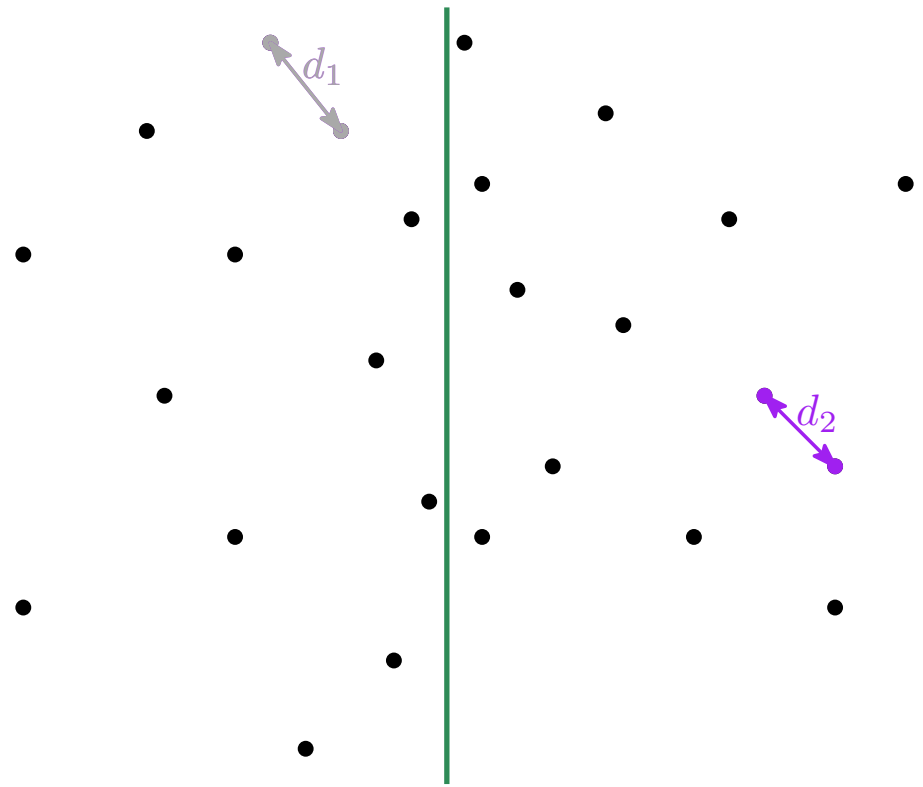
Partitionnement `gauche` – `droite`

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

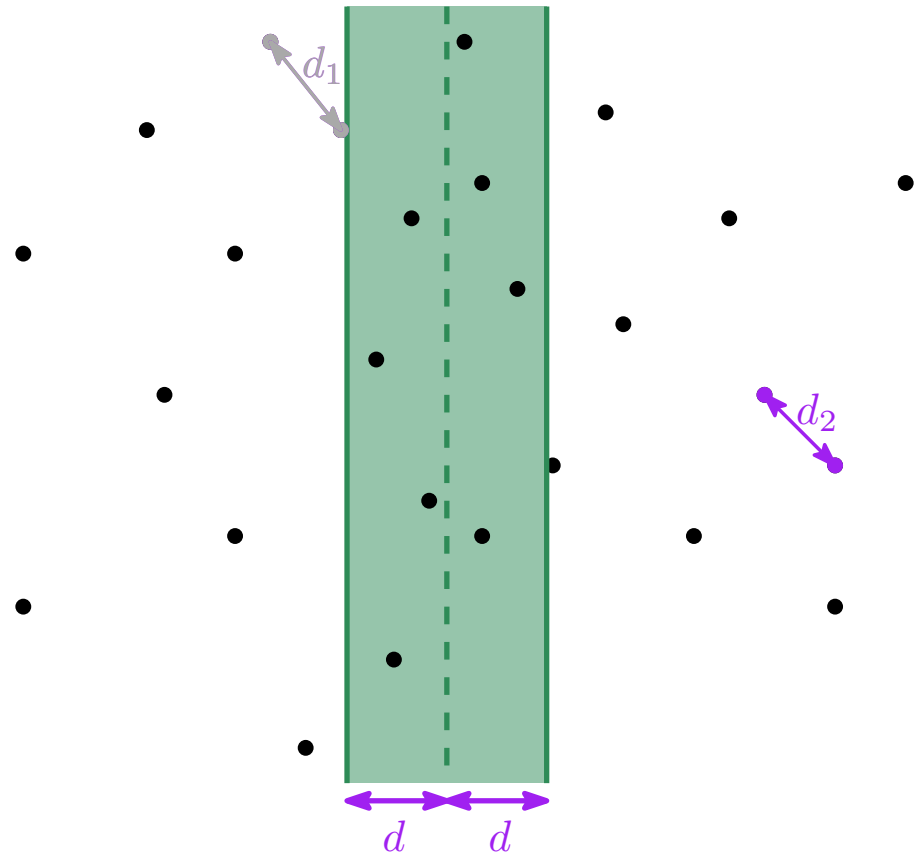


Appels récursifs sur `gauche` et `droite`

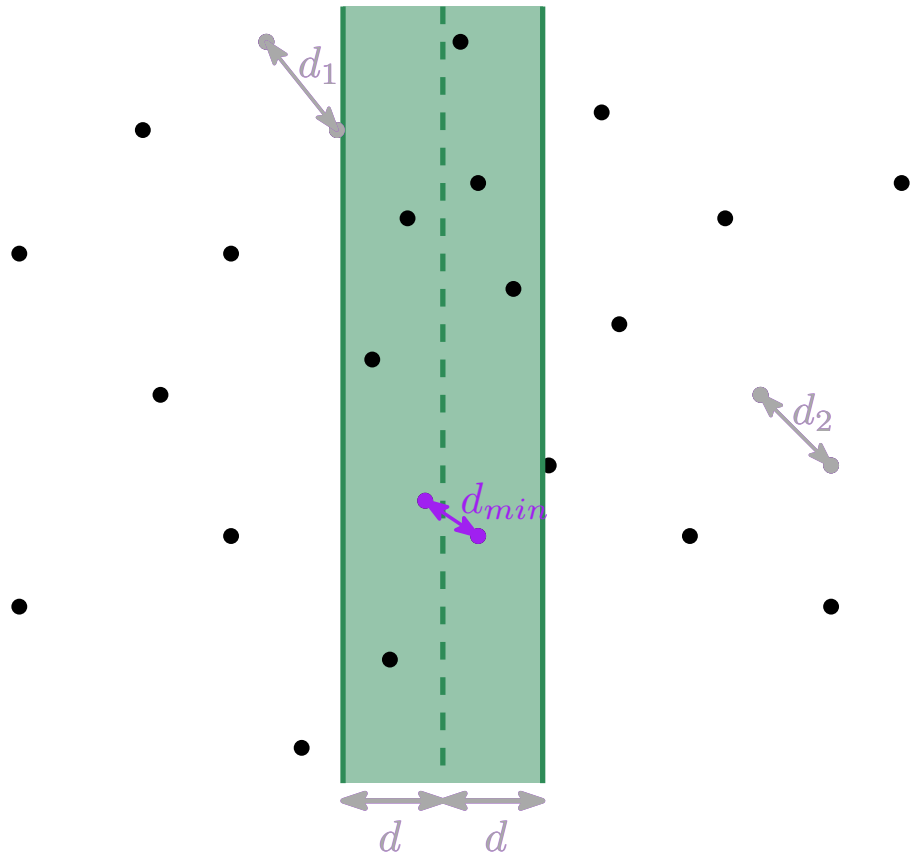
Points les plus proches – *« diviser pour régner »*



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

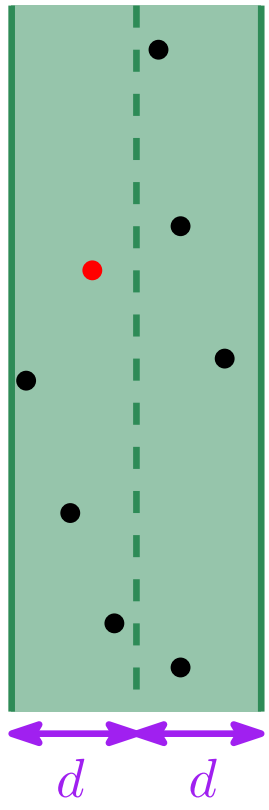


Extraction de la bande médiane de largeur 2d

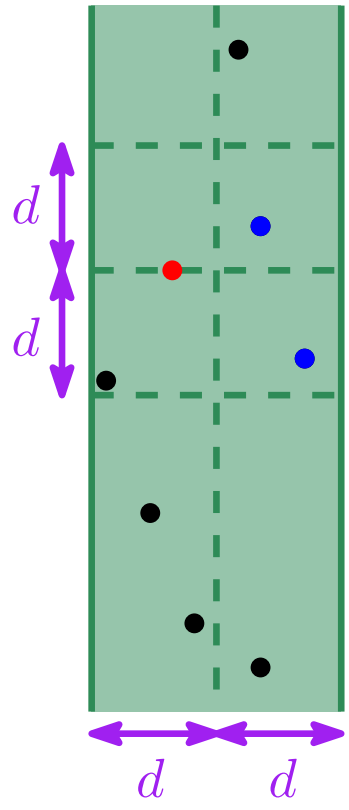


Recherche dans la bande médiane

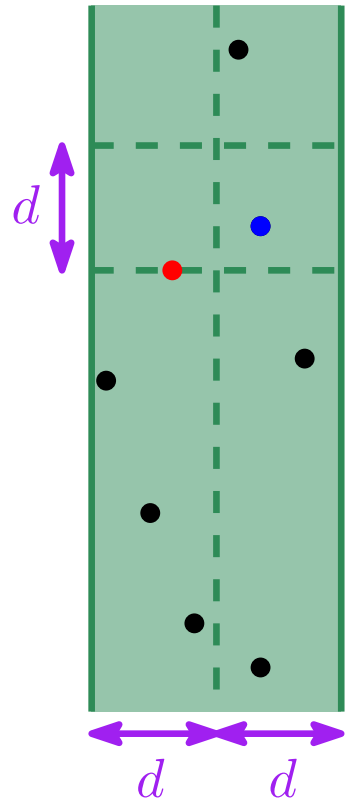
Comment trouver (`p1`, `p2`) efficacement ?



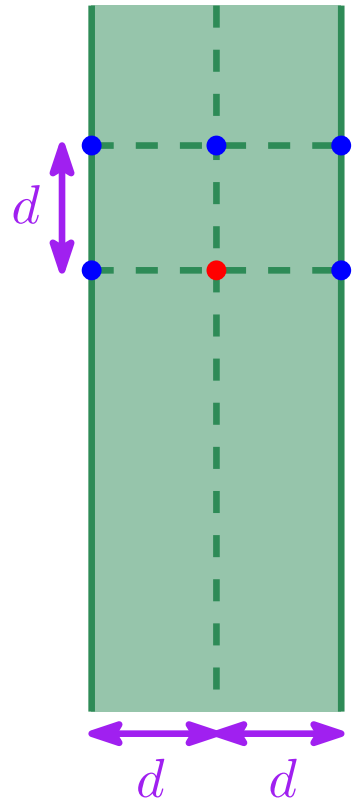
Comment trouver (`p1`, `p2`) efficacement ?



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

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

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

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