Cours 6 : Algorithmes classiques

(Élements de complexité)

Recherche

Objectif : trouver un élément X dans une collection Y.

Algorithme :

Recherche

Objectif : trouver un élément X dans une collection Y.

Algorithme :

1) Vérifier si X est l'élément courant de Y

Recherche

Objectif : trouver un élément X dans une collection Y.

Algorithme :

1) Vérifier si X est l'élément courant de Y
2) Si oui : terminer, succès

Recherche

Objectif : trouver un élément X dans une collection Y.

Algorithme :

1) Vérifier si X est l'élément courant de Y
2) Si oui : terminer, succès
3) Si non : si l'object courant est le dernier de Y terminer, echec

Recherche

Objectif : trouver un élément X dans une collection Y.

Algorithme :

1) Vérifier si X est l'élément courant de Y
2) Si oui : terminer, succès
3) Si non : si l'object courant est le dernier de Y terminer, echec
4) Si non : retour à l'étape 1) en déplaçant l'élément courant sur son successeur

Recherche - version 1

Objectif : trouver un élément X dans une collection Y.

```
def  recherche(seq,      truc)   :
     i=0
     long=len(seq)
     while   (i<long)    and  (seq[i]!=truc):
          i+=1
     return   i<long
```

Recherche - version 1

Objectif : trouver un élément X dans une collection Y.

```
def recherche(seq, truc) :
    i=0
    long=len(seq)
    while (i<long) and (seq[i]!=truc):
        i+=1
    return i<long
```

La fonction recherche renvoie True lorsque i<long : la boucle while a été interrompue avant la fin de la séquence

Recherche - exercice

Utiliser l'instruction *for* pour effectuer une recherche, en interrompant le parcours si l'élément est trouvé.

Recherche - exercice

Utiliser l'instruction `for` pour effectuer une recherche, en interrompant le parcours si l'élément est trouvé.

```
def recherche(seq,truc):
    for cour in seq:
        if cour == truc :
            break
    else: return False
    return True
```

Recherche - version 2

Variante utilisant un chien de garde (watchdog)

```
def recherche2(liste, truc) :
    i=0
    liste2=liste+[truc]
    while (liste2[i]!=truc):    i+=1
    return  i<len(liste)
```

Recherche - version 2

Variante utilisant un chien de garde (watchdog)

```
def recherche2(liste, truc) :
    i=0
    liste2=liste+[truc]
    while (liste2[i]!=truc):    i+=1
    return  i<len(liste)
```

Complexité :
Liste de n valeurs $\Rightarrow$ n comparaisons
($2 \times n$ pour l'algorithme précédent)
Si 1 comp = 10^{-4} secondes $\Rightarrow 1,5$ minutes pour 10^6 valeurs

Séquence triée

Objectif : trouver un élément X dans une collection Y triée de façon croissante

Algorithme :

Séquence triée

Objectif : trouver un élément X dans une collection Y triée de façon croissante

Algorithme :

1) Vérifier si X est l'élément courant de Y

Séquence triée

Objectif : trouver un élément X dans une collection Y triée de façon croissante

Algorithme :

1) Vérifier si X est l'élément courant de Y
2) Si oui : terminer, succès

Séquence triée

Objectif : trouver un élément X dans une collection Y triée de façon croissante

Algorithme :

1) Vérifier si X est l'élément courant de Y
2) Si oui : terminer, succès
3) Si non : si l'object courant est le dernier de Y terminer, echec

Séquence triée

Objectif : trouver un élément X dans une collection Y triée de façon croissante

Algorithme :

1) Vérifier si X est l'élément courant de Y
2) Si oui : terminer, succès
3) Si non : si l'object courant est le dernier de Y terminer, echec
4) Si non : retour à l'étape 1) en déplaçant l'élément courant sur son successeur sauf si X < Y[cour] (echec)

Recherche - liste triée

```
def recherche3(liste, truc) :
    # On suppose liste tirée croissante
    i=0
    liste2=liste+[truc]
    while (liste2[i]<truc):      i+=1

    if liste2[i]>truc:
        return len(liste)+1     # Convention : échec
    else:
        return i # Convention : succès
```

Recherche - liste triée

```
def recherche3(liste, truc) :
    # On suppose liste tirée croissante
    i=0
    liste2=liste+[truc]
    while (liste2[i]<truc):      i+=1

    if liste2[i]>truc:
        return len(liste)+1     # Convention : échec
    else:
        return i # Convention : succès
```

Liste de n valeurs $\Rightarrow$ n comparaisons (au pire)

n valeurs $\Rightarrow$ $\frac{n}{2}$ comparaisons (en moyenne)

Recherche dichotomique

```
def dicho(L, T) :
    # On suppose la liste tirée croissante
    g=0          # Élément gauche
    d=len(L)     # Élément droit
    while g<=d:
        m=(g+d)/2    # Milieu de [gd]
        if L[m]==T:  return m
        elif T<L[m]  : d=m-1
        else : g=m+1
    return len(L)+1
```

Recherche dichotomique

```
def dicho(L, T) :
    # On suppose liste tirée croissante
    g=0          # Élément gauche
    d=len(L)     # Élément droit
    while g<=d:
        m=(g+d)/2    # Milieu de [gd]
        if L[m]==T:  return m
        elif T<L[m]  : d=m-1
        else : g=m+1
    return len(L)+1
```

Liste de n valeurs $\Rightarrow$ $1+\log_2(n)$ comparaisons (au pire)
n valeurs $\Rightarrow$ $\log_2(n)$ comparaisons (en moyenne)

Comparaison

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^{9} données

Comparaison

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^9 données

Un algorithme linéaire prend 10^6 secondes soit plus de 11 jours

Comparaison

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^{9} données

Un algorithme linéaire prend 10^{6} secondes soit plus de 11 jours

Un algorithme logarithmique prend 3×10^{-2} secondes soit moins d'un dixième de secondes

Comparaison

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^{9} données

Un algorithme linéaire prend 10^{6} secondes soit plus de 11 jours

Un algorithme logarithmique prend 3×10^{-2} secondes soit moins d'un dixième de secondes

Ce types d'algorithme est utilisé pour organiser les DNS
Une recherche linéaire est inutilisable en pratique pour les DNS

Algorithmes de tri

La recherche beaucoup plus efficace si liste triée

Deux options :

Algorithmes de tri

La recherche beaucoup plus efficace si liste triée

Deux options :

Garder la liste triée à chaque nouvel ajout

Algorithmes de tri

La recherche beaucoup plus efficace si liste triée

Deux options :

Garder la liste triée à chaque nouvel ajout

Problèmes : délicat à mettre en œuvre
pas toujours possible

Algorithmes de tri

La recherche beaucoup plus efficace si liste triée

Deux options :

Garder la liste triée à chaque nouvel ajout

Problèmes : délicat à mettre en œuvre
pas toujours possible

Trier la liste

Algorithmes de tri

La recherche beaucoup plus efficace si liste triée

Deux options :

Garder la liste triée à chaque nouvel ajout

Problèmes : délicat à mettre en œuvre
pas toujours possible

Trier la liste

Problème : tri naïf très peu efficace

Tri naïf

(Tri par sélection)

Objectif : transformer une séquence Y de façon à ce que le résultat soit trié en ordre croissant
Algorithme :

Tri naïf

(Tri par sélection)

Objectif : transformer une séquence Y de façon à ce que le résultat soit trié en ordre croissant
Algorithme :

1) Trouver l'élément le plus petit de la séquence Y

Tri naïf

(Tri par sélection)

Objectif : transformer une séquence Y de façon à ce que le résultat soit trié en ordre croissant
Algorithme :

1) Trouver l'élément le plus petit de la séquence Y
2) Le placer au début de la séquence

Tri naïf

(Tri par sélection)

Objectif : transformer une séquence Y de façon à ce que le résultat soit trié en ordre croissant
Algorithme :

1) Trouver l'élément le plus petit de la séquence Y
2) Le placer au début de la séquence
3) retour à l'étape 1) en utilisant comme séquence Y privée de son premier élément (le plus petit)

Tri par sélection

Objectif : trier la liste `L`.

```
def selection (L):
    #Attention L va être modifiée
    i=0
    long=len(L)
    while i<long :
        min,j = i,i+1
        while j<long:
            if L[j]<L[min]: min=j
            j+=1
        L[i],L[min]=L[min],L[i]
        i+=1
    return L
```

Variante : Tri à bulles

Objectif : trier la liste `L`.

```
def bulle (L):
    long = len(L)
    for i in range(long-1):
        for j in range(long-1,i,-1):
            if L[j-1]>L[j]:
                L[j],L[j-1]=L[j-1],L[j]
    return L
```

La liste `L[i:]` est parcourue de droite à gauche

Deux valeurs consécutives sont échangées si `L[j-1]>L[j]`

On recommence sur `L[i+1:]`

Complexité

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^{6} données

Complexité

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^{6} données

Un algorithme linéaire prend 10^{3} secondes soit moins de 2 minutes

Complexité

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^6 données

Un algorithme linéaire prend 10^3 secondes soit moins de 2 minutes

Un algorithme quadratique ($x \rightarrow x^2$) prend 10^9 secondes soit plus de 31 ans

Complexité

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^6 données

Un algorithme linéaire prend 10^3 secondes soit moins de 2 minutes

Un algorithme quadratique ($x \rightarrow x^2$) prend 10^9 secondes soit plus de 31 ans

Les algorithmes de tri naïf sont quadratiques

Tri efficace : insertion dichotomique

Objectif : trier la liste `L`

```
def tri_dicho(L):
    long = len(L)
    for i in range(1,long):
        if L[i]< L[i-1]:
            # Sinon L[i] est bien plac é
            L_g=insertion(L[:i],L[i])
            L=L_g+L[i+1:]
        # note : L[x:y] fournit une
        # liste vide en cas de valeur
        # erron ée, par exemple si y >= long
        # Attention aux valeurs n égatives
    return L
```

Tri efficace : insertion dichotomique

La fonction d'insertion

Objectif : placer `val` en bonne place dans la liste `l`

```
def insertion (l, val):
    # On suppose l tirée croissante
    g=0             # Élément gauche
    d=len(l)-1      # Élément droit
    # On a nécessairement l[g]<=val<=d
    # reste vrai dans la boucle
    while (d-g)>=0:
        # Pas encore trouvé la place de val
        m=(g+d)/2   # Milieu de [gd]
        if val<l[m]  : d=m-1
        else : g=m+1
    return l[:g]+[val]+l[g:]
```

Complexité

L'algorithme d'insertion dichotomique a une complexité en $nlog_2(n)$

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^6 données

Complexité

L'algorithme d'insertion dichotomique a une complexité en $nlog_2(n)$

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^6 données

Un algorithme $nlog_2(n)$ prend 10^4 secondes soit moins de 20 minutes

Complexité

L'algorithme d'insertion dichotomique a une complexité en $nlog_2(n)$

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^6 données

Un algorithme $nlog_2(n)$ prend 10^4 secondes soit moins de 20 minutes

Un algorithme quadratique ($x \rightarrow x^2$) prend 10^9 secondes soit plus de 31 ans

Complexité

L'algorithme d'insertion dichotomique a une complexité en $nlog_2(n)$

Si on considère qu'il faut 10^{-3} seconde par opération

Si on considère un corpus de 10^6 données

Un algorithme $nlog_2(n)$ prend 10^4 secondes soit moins de 20 minutes

Un algorithme quadratique ($x \rightarrow x^2$) prend 10^9 secondes soit plus de 31 ans

Les algorithmes de tri optimaux sont tous en $nlog_2(n)$.

Autres algorithmes de tri

Dans la pratique le tri par insertion dichotomique est peu employé

Autres algorithmes de tri

Dans la pratique le tri par insertion dichotomique est peu employé

Les tris quicksort ou fusion (merge) sont plus employés

Autres algorithmes de tri

Dans la pratique le tri par insertion dichotomique est peu employé

Les tris quicksort ou fusion (merge) sont plus employés

Ils reposent sur la récursivité

Autres algorithmes de tri

Dans la pratique le tri par insertion dichotomique est peu employé

Les tris quicksort ou fusion (merge) sont plus employés

Ils reposent sur la récursivité

Ils seront vu au second semestre