Partie 5

Tri et recherche

11 novembre 2019

Plan

1. Recherche
2. Tri
3. Application aux problèmes 2SUM et 3SUM
4. Diviser pour régner

Introduction

Les algorithmes de recherche et de tri sont des algorithmes importants en informatique.

Ils sont directement utiles mais aussi à la base de nombreux autres algorithmes.

Objectifs de cette leçon :

- Vous présenter des solutions efficaces à ces deux problèmes : recherche dichotomique et tri par fusion
- Vous convaincre de l'importance de développer des solutions efficaces
- Vous montrer que le tri et la recherche permet de résoudre efficacement d'autres problèmes algorithmiques.
- Vous présenter la technique du "diviser-pour-régner".

Illustration : filtrage d'adresses emails

On gère un serveur d'emails et on aimerait ajouter une fonctionalité de filtrage des adresses, soit :

- Liste noire : on ne veut pas laisser passer les emails des personnes de la liste
- Liste blanche : on ne veut laisser passer que les emails des personnes de la liste.

Combien d'emails peut-on espérer filtrer à la seconde en fonction de la longueur de la liste ?

Recherche linéaire : tableau quelconque

Warm-up : recherche dans un tableau d'entiers :

- Soit un tableau d'entiers, on veut déterminer si une valeur `key` se trouve dans le tableau.

Solution naïve sans faire d'hypothèse sur les valeurs du tableau :

```
int linear_search(int key, int tab[], int n) {
  for (int i = 0; i < n; i++) {
    if (tab[i] == key)
       return i;
  }
  return -1;
}
```

Complexité : $O(n)$ pour un tableau de taille n.

- Pire cas : l'entier recherché n'est pas dans la liste.

Recherche linéaire : tableau trié

Si on suppose que le tableau est trié, on peut s'arrêter plus tôt dans la recherche.

```
int sorted_linear_search(int key, int tab[], int n) {
  int i = 0;
  while (i < n && key > tab[i])
     i++;

  if (tab[i] == key)
     return i;
  else
     return -1;
}
```

Complexité identique : $O(n)$ pour un tableau de taille n.

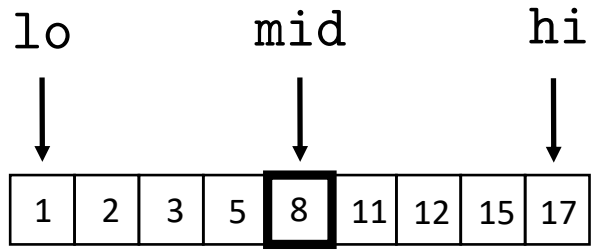
- Pire cas : l'entier recherché est plus grand que toutes les valeurs dans le tableau

Recherche dichotomique (*binary search*)

On peut faire (beaucoup) mieux si on suppose que le tableau est trié.

Idée : On compare la valeur recherchée à la valeur au milieu du tableau :

- Si elle est égale, on la renvoie
- Si elle est plus petite, on recherche récursivement la valeur dans la première moitié du tableau
- Si elle est plus grande, on recherche récursivement la valeur dans la seconde moitié du tableau



Recherche dichotomique : implémentation récursive

```
int binary_search_aux(int key, int tab[], int lo, int hi) {

  if (lo > hi) return -1;

  int mid = lo + (hi - lo) / 2;

  if (key == tab[mid])
    return mid;
  else if (key < tab[mid])
    return binary_search_aux(key, tab, lo, mid-1);
  else
    return binary_search_aux(key, tab, mid+1, hi);
}

int binary_search(int key, int tab[], int n) {
  return binary_search_aux(key, tab, 0, n-1);
}
```

Remarque : `lo+(hi-lo)/2` est préférable à `(lo+hi)/2` pour éviter un dépassement de valeur si `lo` est un entier très grand.

Analyse de complexité

Le nombre d'appels récursifs est maximum lorsque la valeur ne se trouve pas dans le tableau.

Supposons pour simplifier les calculs que la taille du tableau n soit telle que $n = 2^k - 1$ pour un entier $k > 0$ ($\Rightarrow k = \log_2(n+1)$).

Les tailles des sous-tableaux considérés à chaque étape sont :

$$2^k - 1, 2^{k-1} - 1, 2^{k-2} - 1, \ldots, 2^1 - 1, 2^0 - 1$$

Il faudra donc $k + 1$ appels récursifs avant d'arriver au cas de base (`lo>hi`).

En dehors de l'appel récursif, le corps de la fonction est $O(1)$.

La complexité en temps est donc $O(\log n)$.

La complexité en espace est aussi $O(\log n)$ (au plus k appels récursifs imbriqués).

Recherche dichotomique : implémentation itérative

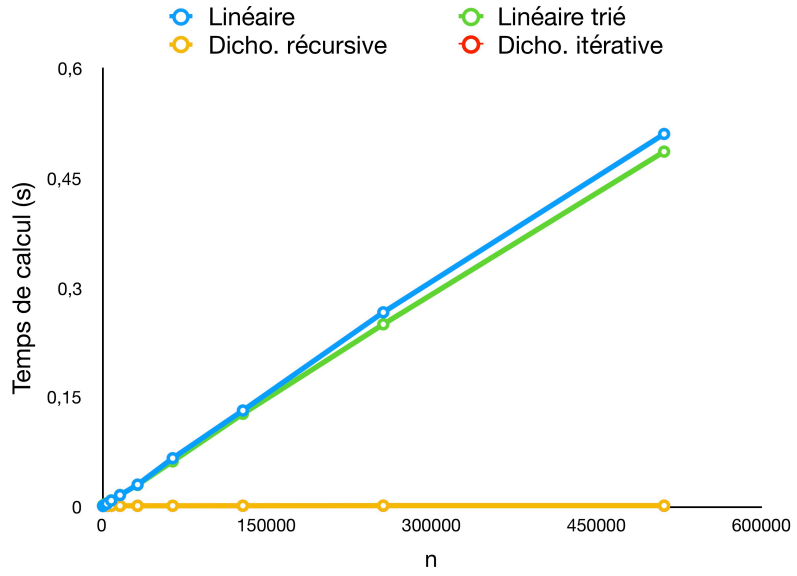
```
int binary_search_iter(int key, int tab[], int n) {
  int lo = 0;
  int hi = n-1;

  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    if (key < tab[mid])
      hi = mid - 1;
    else if (key > tab[mid])
      lo = mid + 1;
    else
      return mid;
  }

  return -1;
}
```

Complexité en temps : $O(\log n)$
Complexité en espace : $O(1)$

Analyse des temps de calcul

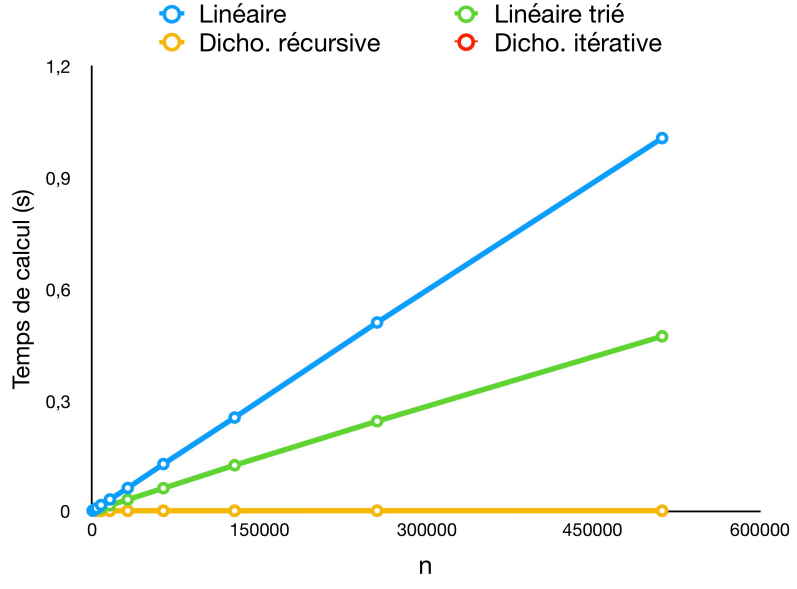
Recherche positive :

- Tableau $[0, 1, \ldots, n-1]$
- 1000 recherches d'une clé : `rand()%n`



Recherche négative :

- Tableau $[0, 2, 4, \ldots, 2n-2]$
- 1000 recherches d'une clé : `rand()%(2*n)+1`



Filtrage d'adresses email : implémentation

Adaptation du code à la recherche d'une chaîne de caractères.

```
int binary_search_iter(char *key, char **tab, int n) {
  int lo = 0;
  int hi = n-1;

  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    int cmp = strcmp(key, tab[mid]);
    if (cmp < 0)
      hi = mid - 1;
    else if (cmp > 0)
      lo = mid + 1;
    else
      return mid;
  }

  return -1;
}
```

Remarques :

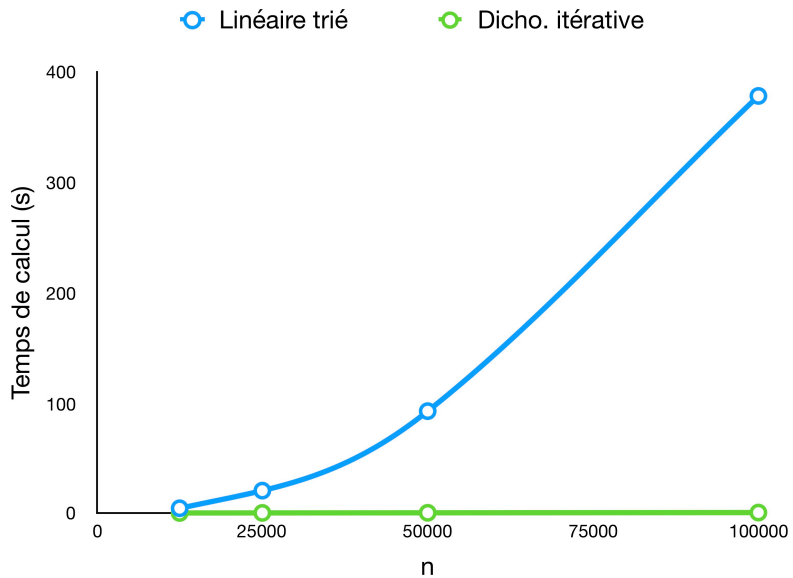
- `strcmp(s1,s2)` renvoie 0 si les deux chaînes sont identiques ou un entier < 0 (resp. > 0) si `s1` est avant (resp. après) `s2` dans l'ordre lexicographique (`string.h`).
- On peut aussi écrire une fonction générique en se basant sur des pointeurs sur void et de fonctions (cf. partie 3).

Filtrage d'adresses email : temps de calcul

Génération de données :

- Liste blanche : n chaînes de caractères (a–z) aléatoires de longueur 10.
- Requêtes : $10n$ chaînes prises au hasard dans la liste (que des recherches positives)

n	Rech. lin.(s)	Rech. dicho.(s)
12500	4,30	0,03
25000	20,31	0,08
50000	92,41	0,19
100000	378,42	0,39



Pour une table de $n = 100000$ adresses email :

- Recherche dichotomique : 256000 vérifications par seconde.
- Recherche linéaire : 264 vérifications par seconde.

Recherche en $O(1)$

Supposons que le tableau ne contienne que des valeurs entières positives codées sur 8 bits ($0 \leq \texttt{key} < 256$).

On peut représenter le tableau par une vecteur de taille 256 dont la $i$ème valeur vaut 1 si i appartient au tableau, 0 sinon.

Fonction de recherche sous ces hypothèses :

```
int constant_search(int key, unsigned char tab[]) {
  return tab[key];
}
```

Complexité en temps : $O(1)$

Limitations évidentes :

- Ne marche que lorsque les valeurs du tableau sont des entiers bornés.
- Augmente l'espace mémoire nécessaire si l'ensemble de valeurs est petit.

Plan

1. Recherche

2. Tri

3. Application aux problèmes 2SUM et 3SUM

4. Diviser pour régner

Tri

Un des problèmes algorithmiques les plus fondamentaux.

Applications innombrables : tri des mails selon leur ancienneté, tri des résultats de requêtes sur Google, tri des facettes des objets pour l'affichage 3D, gestion des opérations bancaires...

Sert de brique de base pour de nombreux autres algorithmes

- Recherche dichotomique
- Recherche des éléments dupliqués dans une liste
- Recherche du $k$ème élément le plus grand dans une liste
- 3SUM
- ...

Environ 25% du temps de calcul des ordinateurs est utilisé pour trier.

Deux algorithmes quadratiques : tri par sélection

Idée : on ramène itérativement le minimum du reste du tableau à la position courante.

```
void selectionsort(int tab[], int n) {
  for (int i = 0; i < n-1; i++) {
    int imin = i;
    int j;
    for (j = i+1; j < n; j++) {
      if (tab[j] < tab[imin])
        imin = j;
    }
    if (imin != j) {
      int tmp = tab[i];
      tab[i] = tab[imin];
      tab[imin] = tmp;
    }
  }
}
```

Complexité : $O(n^2)$

- Double boucle complète quel que soit le contenu du tableau

Deux algorithmes quadratiques : tri par insertion

Idée : on insére la valeur à la position i à sa bonne position dans le sous-tableau qui la précéde supposé préalablement trié.

```
void insertionsort(int tab[], int n) {
  int i = 1;
  while (i < n) {
    int key = tab[i];
    int j = i;
    while (j > 0 && tab[j-1]>key) {
      tab[j] = tab[j-1];
      j--;
    }
    tab[j] = key;
    i++;
  }
}
```

Complexité : $O(n^2)$

- Pire cas : la valeur `key` doit être ramenée au début du tableau à chaque itération de la boucle externe $\Rightarrow$ tableau trié par ordre décroissant.
- Plus efficace que le tri par sélection sur des tableaux presque triés.

Un tri plus efficace

Les tris par sélection ou par insertion sont trop lents pour des applications à grande échelle (voir tests plus loin).

On peut faire (beaucoup) mieux en se basant sur une approche récursive.

Idée du tri par fusion :

- Diviser le tableau en deux.
- Trier les deux sous-tableaux récursivement.
- Fusionner les deux sous-tableaux triés.

Inventé par John von Neumann en 1945, un mathématicien ayant conçu l'architecture des premiers ordinateurs modernes.

Tri par fusion : fonction principale

```
void mergesort(int tab[], int n) {
  mergesort_aux(0, tab, 0, n-1);
}

static void mergesort_aux(int tab[], int lo, int hi) {
  int n = hi - lo + 1;
  if (n <= 1)
    return;
  int mid = lo + (n + 1) / 2;
  mergesort_aux(tab, lo, mid - 1);
  mergesort_aux(tab, mid, hi);
  merge(tab, lo, mid, hi); // fusionne les sous-tableaux triés
                           // tab[lo..mid-1] et tab[mid..hi]
}
```

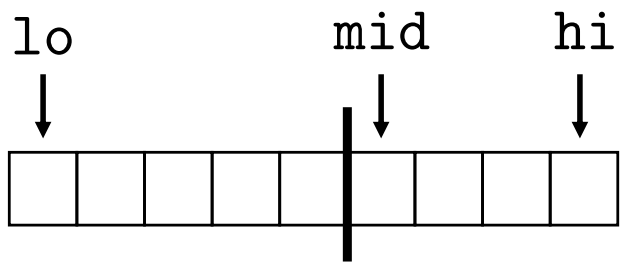
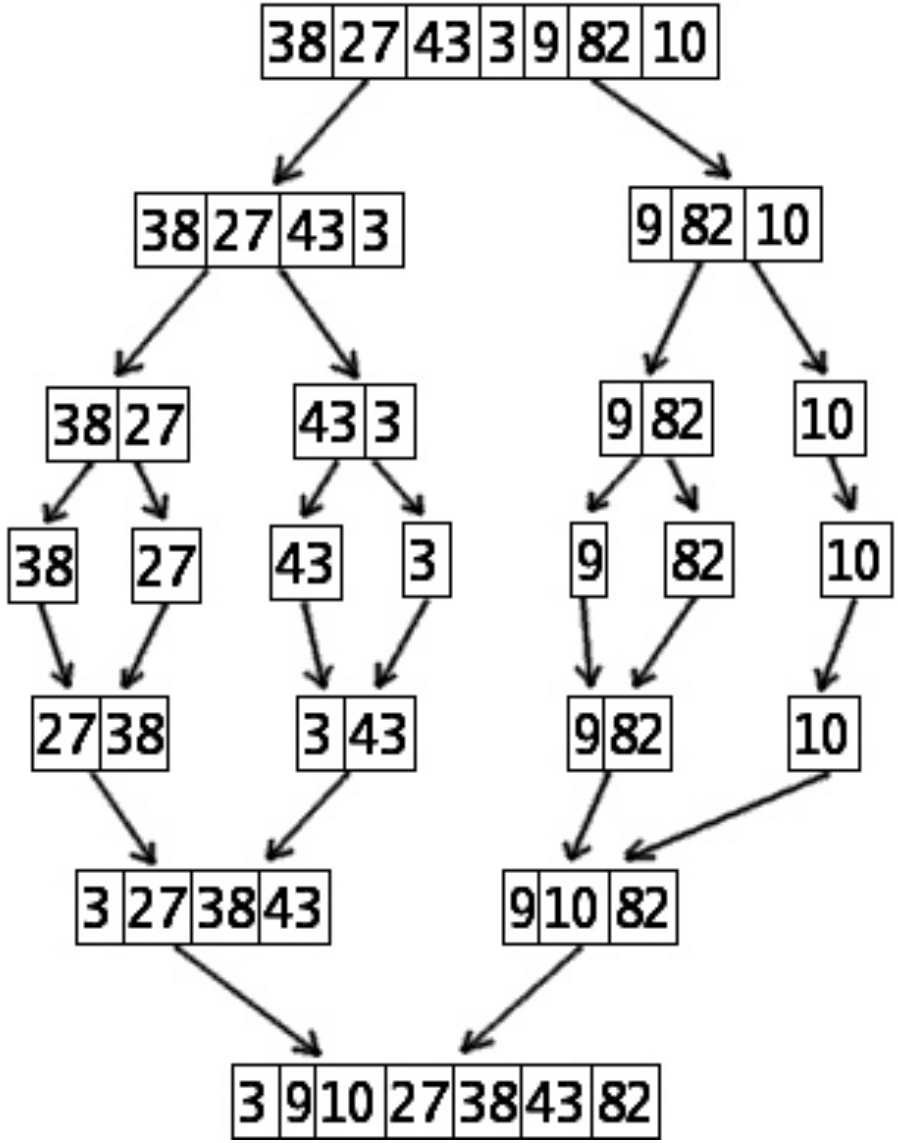


Illustration : trace des appels récursifs

```
mergesort_aux(lo=0, hi=6)
  |mergesort_aux(lo=0, hi=3)
  |  |mergesort_aux(lo=0, hi=1)
  |  |  |mergesort_aux(lo=0, hi=0)
  |  |  |mergesort_aux(lo=1, hi=1)
  |  |  |merge(lo=0, mid=1, hi=1)
  |  |mergesort_aux(lo=2, hi=3)
  |  |  |mergesort_aux(lo=2, hi=2)
  |  |  |mergesort_aux(lo=3, hi=3)
  |  |  |merge(lo=2, mid=3, hi=3)
  |  |merge(lo=0, mid=2, hi=3)
  |mergesort_aux(lo=4, hi=6)
  |  |mergesort_aux(lo=4, hi=5)
  |  |  |mergesort_aux(lo=4, hi=4)
  |  |  |mergesort_aux(lo=5, hi=5)
  |  |  |merge(lo=4, mid=5, hi=5)
  |  |mergesort_aux(lo=6, hi=6)
  |  |merge(lo=4, mid=6, hi=6)
  |merge(lo=0, mid=4, hi=6)
```

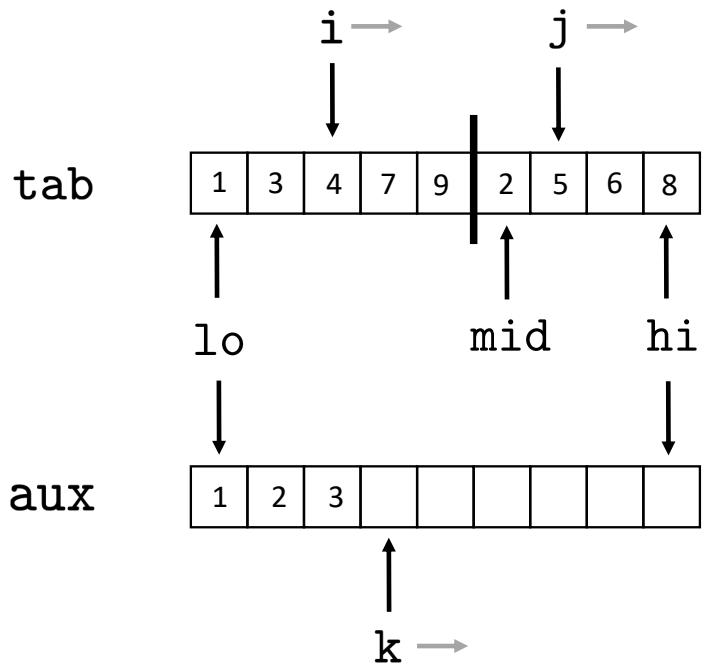


Source : wikipedia

Fusion

Idée (en utilisant un tableau auxiliaire) :

- Un indice pointe vers le début de chacun des deux sous-tableaux.
- On recopie la valeur la plus petite pointée dans le tableau auxiliaire, on incrémente son indice et on recommence jusqu'à ce que toutes les valeurs soient copiées.
- Le tableau auxiliaire est recopié dans le tableau de départ.



Fusion : implémentation

```
static void merge(int tab[], int lo, int mid, int hi, int aux[]) {

  int i = lo, j = mid;

  for (int k = lo; k <= hi; k++)
    if (i == mid)
      aux[k] = tab[j++];
    else if (j == hi + 1)
      aux[k] = tab[i++];
    else if (tab[i] < tab[j])
      aux[k] = tab[i++];
    else
      aux[k] = tab[j++];

  for (int k = lo; k <= hi; k++)
    tab[k] = aux[k];

}
```

Complexité : $O(n)$ avec $n = \texttt{hi} - \texttt{lo} + 1$ la taille totale des deux sous-tableaux.

Tri par fusion : code complet

```
void mergesort(int tab[], int n) {
  int aux[n];
  mergesort_aux(0, tab, 0, n-1, aux);
}

static void mergesort_aux(int tab[], int lo, int hi, int aux[]) {
  int n = hi - lo + 1;
  if (n <= 1)
    return;
  int mid = lo + (n + 1) / 2;
  mergesort_aux(tab, lo, mid - 1, aux);
  mergesort_aux(tab, mid, hi, aux);
  merge(tab, lo, mid, hi, aux);
}

static void merge(int tab[], int lo, int mid, int hi, int aux[]) {
  int i = lo, j = mid;
  for (int k = lo; k <= hi; k++)
    if (i == mid)
      aux[k] = tab[j++];
    else if (j == hi + 1)
      aux[k] = tab[i++];
    else if (tab[i] < tab[j])
      aux[k] = tab[i++];
    else
      aux[k] = tab[j++];

  for (int k = lo; k <= hi; k++)
    tab[k] = aux[k];
}
```

Analyse de complexité

Supposons pour simplifier les calculs que la taille du tableau n soit telle que $n = 2^k$ pour un entier $k > 0$ ($\Rightarrow k = \log_2 n$).

On a les appels suivants de la fonction `merge` :

- 1 appel sur un tableau de taille n $\Rightarrow O(n)$
- 2 appels sur des sous-tableaux de tailles $n/2$ $\Rightarrow O(n)$
- 4 appels sur des sous-tableaux de tailles $n/4$ $\Rightarrow O(n)$
- ... ...
- $n/2$ appels sur des sous-tableaux de taille 2 $\Rightarrow O(n)$

On a donc au total $k = \log_2 n$ opérations de complexité $O(n)$.

La complexité en temps est $O(n \log n)$.

La complexité en espace est $O(n + \log n) = O(n)$.

- $O(n)$ pour le tableau auxiliaire, $O(\log n)$ pour les appels récursifs.

Remarques

- Les temps de calcul ne dépendent pas du contenu du tableau, contrairement au tri par insertion.
- Il est possible d'implémenter l'algorithme itérativement et/ou sans utiliser de tableau auxiliaire mais c'est plus compliqué.
- On peut montrer qu'il n'est pas possible d'écrire un algorithme de tri meilleur que $O(n \log n)$, sans faire d'hypothèse supplémentaire sur la nature des valeurs à trier (cf INFO0902).

Filtrage d'adresses email

Adaptation de la fonction merge pour le tri de chaînes de caractères :

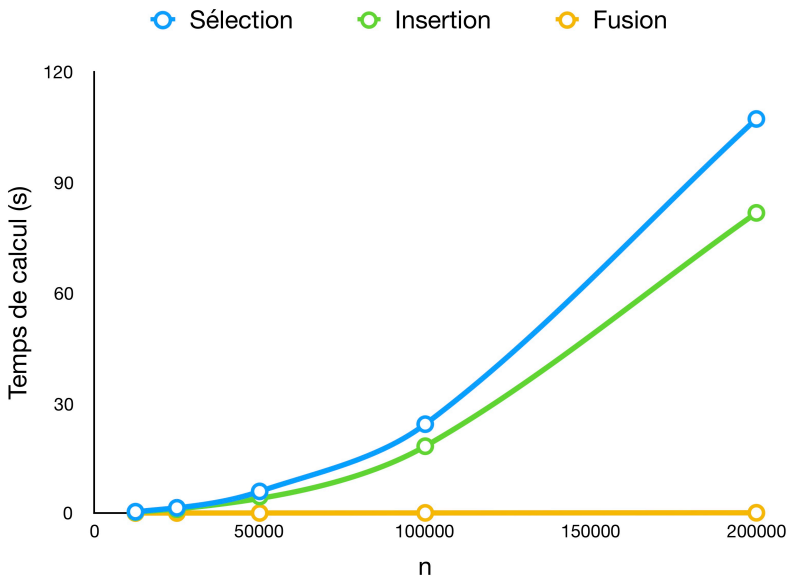
```
static void merge_str(char **tab, int lo, int mid, int hi, char **aux) {
  int i = lo, j = mid;

  for (int k = lo; k <= hi; k++)
    if (i == mid)
      aux[k] = tab[j++];
    else if (j == hi + 1)
      aux[k] = tab[i++];
    else if (strcmp(tab[i], tab[j]) < 0)
      aux[k] = tab[i++];
    else
      aux[k] = tab[j++];

  for (int k = lo; k <= hi; k++)
    tab[k] = aux[k];

}
```

Les autres fonctions peuvent être adaptées trivialement (en changeant `int []` en `char **`).

Analyse empirique

Génération de données : n chaînes de caractères aléatoires de longueur 10.

n	Sélection (s)	Insertion (s)	Fusion (s)
12500	0,36	0,22	$<$0,01
25000	1,42	0,99	$<$0,01
50000	5,88	4,48	0,01
100000	24,18	18,18	0,03
200000	107,26	81,69	0,06



Pour 1 millions d'adresses emails :

- Tri par sélection : 44 minutes
- Tri par insertion : 33 minutes
- Tri par fusion : 0,3 secondes

Passer de $O(n^2)$ à $O(n \log n)$ fait une énorme différence.

1. Recherche

2. Tri

3. Application aux problèmes 2SUM et 3SUM

4. Diviser pour régner

Problème 2SUM

Étant donné un tableau de n entiers uniques, trouver (ou compter) les paires d'entiers qui somment à 0.

Solution naïve :

```
int count_2sum(int tab[], int n) {
  int count = 0;
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
        if (tab[i] + tab[j] == 0)
          count++;
  return count;
}
```

Complexité en temps : $O(n^2)$

Problème 2SUM : 1ère solution basée sur le tri

Une première solution basée sur le tri et la recherche dichotomique :

- On trie le tableau (en utilisant le tri par fusion).
- Pour chaque élément `tab[i]` (négatif), on recherche `-tab[i]` dans le reste du tableau en utilisant la recherche dichotomique.

```
int count_2sum_bs(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int i = 0;
  while (i < n-1 && tab[i] < 0) {
    if (binary_search(-tab[i],tab+i+1,n-i) != -1)
      count++;
    i++;
  }
  return count;
}
```

Problème 2SUM : 1ère solution basée sur le tri

```
int count_2sum_bs(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int i = 0;
  while (i < n-1 && tab[i] < 0) {
    if (binary_search(-tab[i],tab+i+1,n-i) != -1)
      count++;
    i++;
  }
  return count;
}
```

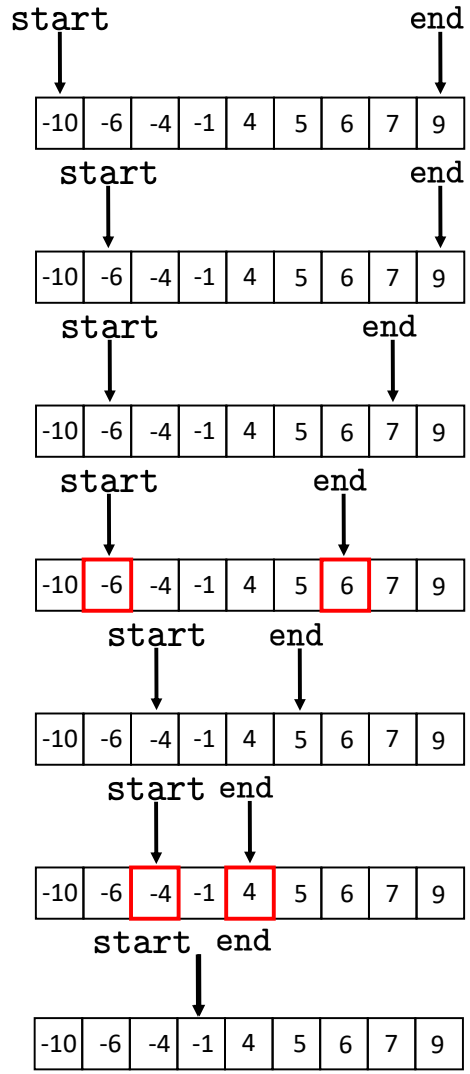
Complexité en temps : $O(n \log n)$

- Tri par fusion : $O(n \log n)$
- Pire cas pour la boucle `for` : toutes les valeurs sont négatives.
  - n recherches dichotomiques en $O(\log n) \Rightarrow O(n \log n)$.

Problème 2SUM : 2ème solution basée sur le tri

On peut se passer de la recherche dichotomique.

```
int count_2sum_fast(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int start = 0;
  int end = n-1;
  while (start < end) {
    int sum = tab[start] + tab[end];
    if (sum == 0) {
      count++;
      start++;
      end--;
    } else if (sum > 0)
      end--;
    else
      start++;
  }
  return count;
}
```



Problème 2SUM : 2ème solution basée sur le tri

```
int count_2sum_fast(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int start = 0;
  int end = n-1;
  while (start < end) {
    int sum = tab[start] + tab[end];
    if (sum == 0) {
      count++;
      start++;
      end--;
    } else if (sum > 0)
      end--;
    else
      start++;
  }
  return count;
}
```

Complexité en temps identique à la 1ère solution : $O(n \log n)$

- Tri par fusion : $O(n \log n)$
- Boucle `while` : $O(n)$, négligeable par rapport au tri

Problème 3SUM : 1ère solution basée sur le tri

En se basant sur la recherche dichotomique :

```
int count_3sum_bs(int tab[], int n) {
  mymergesort(tab,n);
  int count = 0;
  int i = 0;
  while (i < n-2 && tab[i] < 0) {
    int j = i+1;
    while (j < n-1 && tab[i]+tab[j] < 0) {
      if (binary_search(-(tab[i]+tab[j]), tab+j+1, n-j) != -1)
        count++;
      j++;
    }
    i++;
  }
  return count;
}
```

Complexité : $O(n^2 \log n)$

- Double boucle : $O(n^2)$ itérations et recherche dichotomique en $O(\log n) \Rightarrow O(n^2 \log n)$
- Tri par fusion, $O(n \log n)$, négligeable

Problème 3SUM : 2ème solution basée sur le tri

```
int count_3sum_fast(int tab[], int n) {
  mergesort(tab,n);
  int count = 0;
  int i = 0;
  while (i < n-1 && tab[i] < 0) {
    int start = i + 1;
    int end = n - 1;
    while (start < end) {
      int sum = tab[i] + tab[start] + tab[end];
      if (sum == 0) {
        count++; start++; end--;
      } else if (sum > 0)
        end--;
      else
        start++;
    }
    i++;
  }
  return count;
}
```

Complexité en temps [3] : $O(n^2)$

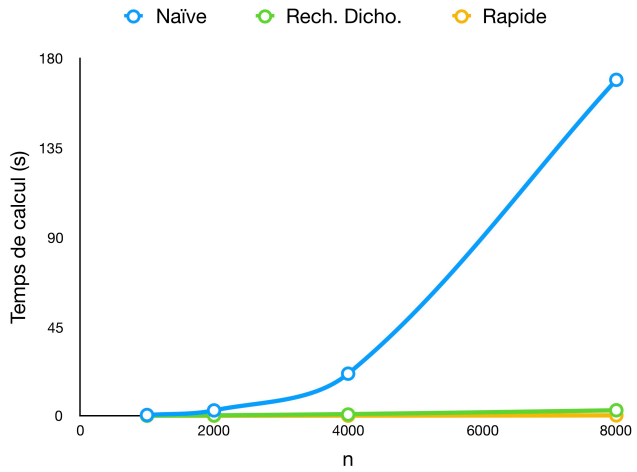
- Double boucle : $O(n^2)$ itérations
- Tri par fusion, $O(n \log n)$, négligeable

3. Longtemps considérée comme la solution optimale mais une solution $O(n^2/(\log n/\log\log n)^{2/3})$ a été proposée en 2014.

Analyse empirique

Tableaux d'entiers compris entre -1000000 et +999999 sans doublons.

n	Naïve	Rech. bin.	Rapide
1000	0,36	0,03	$< 0,01$
2000	2,68	0,14	0,01
4000	21,20	0,60	0,04
8000	169,29	2,75	0,18



(Sur un Intel Core i7 2.5 Ghz)

Prédiction pour $n = 1000000$:

- $>$ 10 ans pour la version naïve $O(n^3)$ (cf. partie 4)
- 17 heures pour la version utilisant la recherche dichotomique $O(n^2 \log n)$
- 1 heure pour la version rapide $O(n^2)$

Synthèse

- Le tri est un composant essentiel dans beaucoup d'applications.
- Le tri par fusion offre une solution optimale au problème.
- Passer de $O(n^2)$ à $O(n \log n)$ ou de $O(n)$ à $O(\log n)$ fait une énorme différence dans les applications pratiques.
- La recherche dichotomique et le tri par fusion sont basés sur l'idée générale du "diviser-pour-régner", qui permet d'obtenir des solutions efficaces à beaucoup de problèmes.

Plan

1. Recherche

2. Tri

3. Application aux problèmes 2SUM et 3SUM

4. Diviser pour régner

Diviser pour régner

Technique générique de résolution de problèmes basée sur la récursivité.

Principe général : Soit un problème algorithmique à résoudre de taille n

- Si le problème est trivial (typiquement, n est petit), on le résout directement
- Sinon :
  - On divise le problème en b sous-problèmes de tailles n/a (avec $b \geq 0$ et $a > 1$ deux entiers).
  - On résout récursivement ces sous-problèmes
  - On fusionne les solutions aux sous-problèmes pour produire une solution au problème original

Exemples : recherche dichotomique, tri par fusion, calcul de puissance récursif.

Diviser pour régner

La complexité d'une solution diviser pour régner s'écrit :

$$T(n) = bT(n/a) + f(n)$$

où $f(n)$ est la complexité de l'opération de fusion.

D'autant plus efficaces que $f(n)$ est faible, a est grand, et b est petit.

Sans intérêt si $f(n)$ n'est pas strictement meilleur que la complexité d'une solution naïve au problème.

Exemples :

- Recherche dichotomique : $a = 2,\ b = 1,\ f(n) = c \Rightarrow\ T(n) \in O(\log n)$
- Tri par fusion : $a = 2,\ b = 1,\ f(n) = cn \Rightarrow\ T(n) \in O(n \log n)$
- Calcul d'une puissance entière :
  $a = 2,\ b = 1,\ f(n) = c \Rightarrow\ T(n) \in O(\log n)$

Un autre exemple : recherche de pics

- Soit un tableau $tab[0\,..\,n+1]$ de $n+2$ valeurs réelles. On supposera que $tab[0] = tab[n+1] = -\infty$.
- Définition : $tab[i]$, avec $i \in \{1, \ldots, n\}$, est un pic s'il n'est pas plus petit que ses voisins :

$$tab[i-1] \leq tab[i] \geq tab[i+1]$$

  ($tab[i]$ est un maximum local)
- Problème algorithmique : trouver un pic dans le tableau (n'importe lequel)
- Note : il en existe toujours un

Solution naïve

On teste toutes les positions séquentiellement :

```
int findPeak(float *tab, int n) {
  for (int i = 1; i <= n; i++) {
    if ((tab[i-1] <= tab[i]) && (tab[i]<=tab[i+1]))
       return i;
  }
  return -1; // En principe, inutile
}
```

Complexité : $O(n)$

Une solution par diviser-pour-régner

Idée :

- Sonder un élément $tab[i]$ et ses voisins $tab[i-1]$ et $tab[i+1]$
- Si c'est un pic : renvoyer i
- Sinon :
  - les valeurs doivent croître au moins d'un côté

$$tab[i-1] > tab[i] \text{ ou } tab[i] < tab[i+1]$$

  - Si $A[i-1] > A[i]$, on cherche le pic (récursivement) dans $tab[1\,..\,i-1]$
  - Si $A[i+1] > A[i]$, on cherche le pic (récursivement) dans $tab[i+1\,..\,n]$

(Implémentation en C laissée comme exercice.)

Analyse

Est-ce que l'algorithme est correct ? Oui. Il existera toujours un pic du côté choisi (peut se montrer par l'absurde).

Complexité : $a = 2$, $b = 1$, $f(n) = 1 \Rightarrow O(\log n)$ (comme la recherche dichotomique)