CIR2 - Algorithmique Avancée

Sujet TD&P n°2 - Séance n°2

C++. Dichotomies “maison” et d’<algorithm>.

Partie 1 - Introduction

Juste un peu de lecture...

Objectifs. Nous continuons sur les rappels C++ et l’utilisation d’algorithmes de la STL. Ici, nous prenons comme prétexte la recherche dichotomique avec le développement de deux méthodes “maison”, puis l’exécution de celle fournie par la STL. C’est également l’occasion de prendre l’habitude d’utiliser les itérateurs : ils sont nécessaires à la plupart des algorithmes de la STL et sont performants à plusieurs titres lorsqu’ils sont associés à ces mêmes conteneurs de cette même librairie. Enfin, aucun pseudo-code n’est attendu ici, les algorithmes étant fournis.

Rappel. En considérant qu’un tableau d’entiers T est trié, on peut facilement situer une valeur recherchée dans la première ou la seconde moitié du tableau. La recherche dichotomique itère sur ce schéma, c’est-à-dire en divisant par deux l’espace de recherche de la partie restante de T. On peut résumer la méthode avec l’algorithme et la figure qui suivent :

```
Algorithme RechercheDichotomique(
     Entrées    : vecteur, valeurRecherchee, tailleVecteur ;
     Sortie     : index de l’élément trouvé, -1 sinon.)
          min = 1;
          max = tailleVecteur;
          Tant Que min < max Faire
             mid = (min + max) / 2;
             Si vecteur[mid] < valeurRecherchee Alors
                 min = mid + 1;
             Sinon
                 max = mid;
             Fin Si
          Fin Tant Que
          Si vecteur[min] == valeurRecherchee Alors
             Retourner min;
          Sinon
             Retourner -1;
          Fin Si
Fin Algorithme RechercheDichotomique
```


Samuel Deleplanque. (prénom.nom@junia.com)

La figure suivante représente une partie de la trace de l'algorithme sur un tableau de 9 cases où la valeur 9 est recherchée. L'espace de recherche est représenté en bleu. Les variables `min`, `max` et `mid` sont représentées sur 4 itérations de la boucle **Tant Que**.

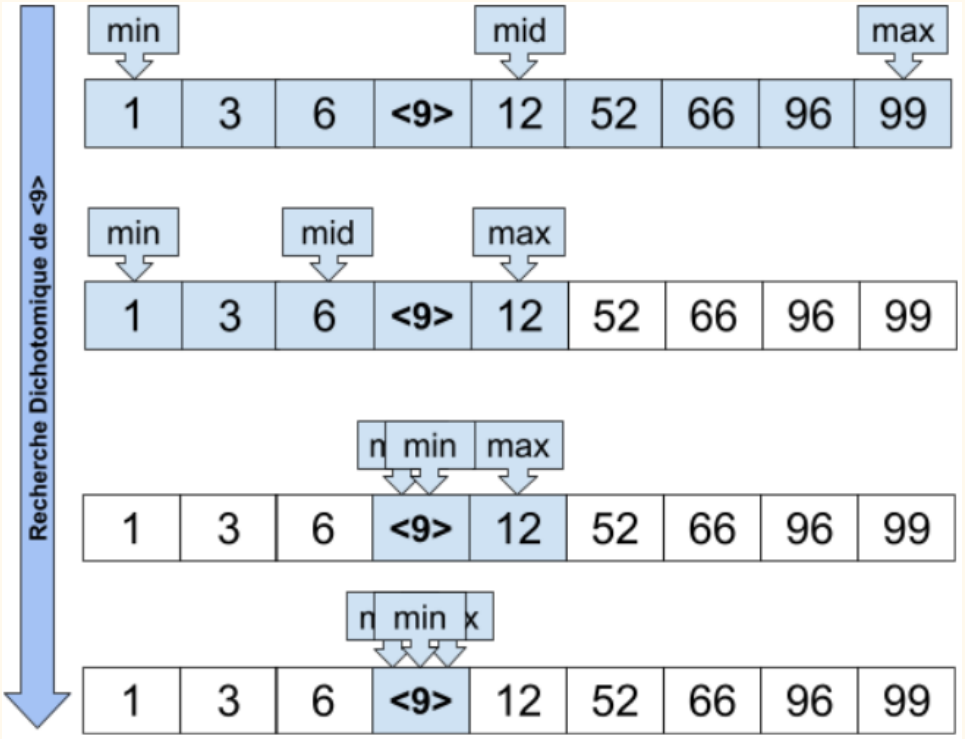

Figure 1 : Trace graphique d'une recherche dichotomique dans un tableau de 9 cases.

Partie 2 - Préparation des données

i. Commencez d'abord par remplir "aléatoirement" le tableau avec des entiers entre 0 et 1000 un `std::array` ou un `std::vector` qui sera la structure où les recherches dichotomiques devront être faites. Les nombres pseudo-aléatoires peuvent s'obtenir avec :

- `rand` (https://en.cppreference.com/w/cpp/numeric/random/rand) mais vous pouvez également utiliser les méthodes vues en cours de C++.

Pensez au **modulo '%'** afin de fixer une borne maximale aux nombres générés, e.g., 1000.

ii. Utilisez ensuite une fonction de tri de `<algorithm>` à appliquer sur le `std::array` :

- `sort` (https://en.cppreference.com/w/cpp/algorithm/sort).

iii. Implémentez en C++ une méthode simple d'affichage de votre vecteur avec une boucle `Pour (For)` utilisant les avantages du C++ "moderne" (i.e., à partir du C++11) que ce soit pour la reconnaissance **auto**matique de type ou en utilisant l'expression la plus courte dans les arguments de la boucle. Il y a quelques exemples dans le cours.

Partie 3 - Développement Algo 1 Maison

iv. Implémentez en C++ l'algorithme de recherche dichotomique écrit plus haut en l'appliquant à votre vecteur. Rendre constant le nombre d'éléments du vecteur (e.g., `constexpr int n = 20;`), ainsi que le nombre à rechercher et également la valeur maximale que chaque entier peut prendre. Pensez à utiliser ces constantes autant que possible.

Partie 4 - Développement Algo 2 Maison & STL

v. Déclarez le ou les itérateurs qui parcourront votre vecteur. Afin d'utiliser quelques algorithmes intéressants de la STL (`#include <algorithm>`)[1]. Essayer de mettre en place une recherche dichotomique dans un `std::vector` en utilisant les algorithmes suivant :

- `distance` (https://en.cppreference.com/w/cpp/iterator/distance). Il donne la "distance" entre deux itérateurs. Utilisez le pour renseigner la taille de la sous-partie dans laquelle il faut poursuivre la recherche dichotomique.

- `advance` (https://en.cppreference.com/w/cpp/iterator/advance). Il va faire bouger l'itérateur : son premier argument prend l'itérateur et le second correspond au nombre de cases à parcourir.

Le code ci-après est à l'image de ce qui doit être fait pour la dichotomie : on calcule la distance, puis en fonction de la position de la valeur recherchée, on fait bouger `iterateurMin` ou `iterateurMax` et systématiquement `interateurMid`.

```
int distance = std::distance (iterateurMin, iterateurMax) / 2;
distance += std::distance (unArrayDejaAlloue.begin (), iterateurMin);
iterateurMid = unArrayDejaAlloue.begin () + distance;
if (*iterateurMid < nombreRecherche)
{
 iterateurMin = iterateurMid;
 std::advance (iterateurMin, 1);
}
else
{
 iterateurMax =  iterateurMid;
}
```

[1] L'objectif principal est la manipulation de ces algorithmes. Il est clair que le contexte de la recherche dichotomique n'est pas optimal pour certains d'entre eux. Par exemple, on pourrait tout à fait faire `iterateurMin++` à la place d'un `std::advance (iterateurMin, 1)`.

- Développez une seconde méthode de dichotomie avec la portion de code précédente. Prenez le temps d'apprendre à bien manipuler les itérateurs si ce n'est pas déjà compris, en particulier pour la condition de sortie de la boucle **Tant Que**. Les grandes lignes de l'algorithme "de base" doivent réapparaître. Enfin, pensez systématiquement à tester les cas d'erreurs et les cas particuliers.

Remarque. Il existe également `std::next` similaire à `std::advance` à la différence près que la première fait des mouvements de proche en proche par défaut alors que `std::advance` requiert une distance en paramètre. Autre différence : `std::next` renvoie un itérateur sur la nouvelle position (sans modifier celui en paramètre) alors que `advance` modifie directement celui en paramètre et "renvoie" `void`.

Partie 5 - Utilisation Algo STL

vi. Utilisez la recherche dichotomique déjà implémentée dans la STL. Cette dernière ne renvoie qu'un booléen indiquant si la valeur recherchée a été trouvée. Vous trouverez plus d'information dans la documentation suivante :

- `std::binary_search` (https://en.cppreference.com/w/cpp/algorithm/binary_search)

Partie 6 - Comparaison des 3 algorithmes

vii. Testez vos trois algorithmes sur le même vecteur initialisé en **Partie 2**.

viii. Augmentez drastiquement la taille de votre vecteur afin de percevoir laquelle des 3 méthodes se démarque par son efficacité. Si vous n'arrivez pas à les différencier, vous pouvez utiliser `chrono` avec `#include <chrono>`, en adaptant l'exemple suivant à votre code :

- *(https://www.geeksforgeeks.org/chrono-in-c/)*.

Aide. Retrouver l'algorithme de recherche dichotomique ainsi qu'une trace pour un tableau donné sur https://www.youtube.com/watch?v=ZrKylsY2p7w (< 3min sans son). Les vidéos peuvent parfois être le meilleur moyen pour comprendre des algorithmes. Plus généralement sur les algorithmes de la STL, vous pouvez vous aider d'une excellente vidéo du `cppcon` 2018 : https://www.youtube.com/watch?v=bFSnXNIsK4A (≈ 55min).

Bonus. Existe-t-il une structure de données arborescente qui permettrait une recherche dichotomique efficace et sans calcul d'indice ? Quelle est la complexité de la recherche dichotomique ? Peut-elle être améliorée par cette autre structure de données, et si oui pourquoi ?

Bon courage.