Informatique *Cours*
Raisonner sur les programmes Ipesup

Objectifs

À l'issue de cette section, l'étudiant devra être capable de :
- ok
- ok
- ok
- ok

Raisonner sur les programmes

Considérons le problème suivant : chercher une éventuelle occurrence d'un élément v dans un tableau a, sachant que les éléments de a sont rangés en ordre croissant. Figurez-vous que la fonction C du programme suivant est une solution à ce problème. Et même, une excellente solution.

Recherche dichotomique

```
1 int binary_search(int v, int a[], int n) {
2     int lo = 0, hi = n;
3     while (lo < hi) {
4         int mid = lo + (hi - lo) / 2;
5         if (a[mid] == v) return mid;
6         if (v < a[mid]) { hi = mid; } else { lo = mid + 1; }
7     }
8     return -1;
9 }
```

Cela n'a toutefois rien d'évident. Une solution plus sûre aurait consisté à énumérer les éléments du tableau dans l'ordre, comme dans la fonction suivante.

```
int sequential_search(int v, int a[], int n) {
    for (int i = 0; i < n; i++) {
        if (a[i] == v) return i;
    }
    return -1;
}
```

Dès lors, comment se convaincre que `binary_search` est effectivement une solution convenable ? Et en quoi peut-elle être meilleure que d'autres solutions comme `sequential_search` ?

Validation par l'expérience.

Pour se convaincre de la fiabilité de la fonction `binary_search`, on pourrait la tester sur quelques exemples et constater que, dans ces cas-là au moins, elle renvoie bien les résultats attendus.

Pour comprendre l'intérêt de `binary_search` par rapport à des alternatives comme `sequential_search`, on peut également mesurer et comparer les performances de ces deux fonctions. Sur l'ordinateur que nous avons utilisé, `binary_search` réalise environ 4 500 000 recherches par seconde dans un tableau de 1 000 000 d'éléments triés en ordre croissant, quand `sequential_search` n'en fait que 700. Manifestement, la première est beaucoup plus rapide que la seconde.

Pouvons-nous prévoir ces résultats, sans recourir à l'expérience ? Pouvons-nous aller plus loin que cette validation empirique ? Est-il possible de démontrer que notre fonction `binary_search` est bien correcte, c'est-à-dire qu'elle fournit à coup sûr un résultat valide, pour l'infinité des combinaisons possibles d'un tableau et d'une valeur cherchée ? Pouvons-nous prédire que `binary_search` allait avoir de meilleures performances que `sequential_search`, et dans quelles proportions ?

Oui. Toutes ces choses sont bien possibles !
Commençons par un tour d'horizon des techniques, appliquées à notre fonction `binary_search`.

Spécification du problème.

Commençons par clarifier le problème auquel `binary_search` doit répondre. Son énoncé contient deux parties :

— les entrées v et a sont un entier et un tableau dont les éléments sont triés par ordre croissant, et n est la taille du tableau a,
— le résultat est l'indice d'une occurrence de v si v apparaît dans a, et -1 sinon.

Le premier point fait partie intégrante de l'énoncé du problème et décrit son périmètre : on ne s'intéresse qu'aux tableaux dont les éléments sont rangés en ordre croissant. Les tableaux mélangés sont hors sujet. Le deuxième point décrit les résultats attendus, en supposant que a respecte bien la première condition. Ces deux points, pris ensemble, spécifient le problème dont on veut montrer que `binary_search` est une solution.

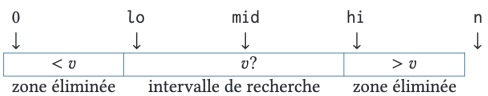
Ainsi, en considérant le tableau a suivant :

$$0 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13 \;\; 21$$

une recherche du nombre 1 peut renvoyer aussi bien 1 que 2, puisque cet élément apparaît deux fois, tandis qu'une recherche de 17 doit renvoyer -1.

Recherche dichotomique.

En prélude à l'analyse, résumons l'action de notre fonction `binary_search` et le rôle que jouent les variables `lo`, `hi` et `mid`.



Les variables `lo` et `hi` définissent un intervalle $[lo, hi[$ du tableau, dans lequel on cherche une éventuelle occurrence de v. Au début de l'algorithme, cet intervalle couvre le tableau entier, puis à chaque étape on en élimine une moitié. Pour cela, on calcule un indice `mid` correspondant au milieu de l'intervalle de recherche, et on compare la valeur cherchée v à la valeur médiane a[`mid`] :

— si v est plus petite, on poursuit la recherche dans la moitié gauche $[lo, \texttt{mid}[$,
— si v est plus grande, on poursuit la recherche dans la moitié droite $[\texttt{mid} + 1, hi[$.

Dans aucun cas on ne conserve `mid` dans l'intervalle de recherche : si la valeur v y est présente, l'algorithme s'arrête de toute façon.

Correction.

Justifions d'abord que tout résultat renvoyé par notre fonction est bien correct, c'est-à-dire conforme aux attendus de la spécification.

1. Lorsque la fonction renvoie un indice avec `return mid`, il s'agit bien d'un indice où se trouve la valeur v : on vient justement de tester ce fait.
2. L'autre résultat possible est -1, renvoyé par l'instruction finale lorsque l'intervalle de recherche devient vide. Il faut donc justifier que dans ce cas, la valeur v ne se trouve effectivement nulle part dans le tableau.

Pour cela on démontre que deux propriétés restent vraies du début à la fin de l'exécution de l'algorithme :

— avant l'indice `lo`, tous les éléments du tableau sont strictement inférieurs à v,
— à partir de l'indice `hi`, tous les éléments sont strictement supérieurs à v.

Ces propriétés, invariantes malgré les évolutions successives de `lo` et `hi`, sont appelées des *invariants*.

La démonstration se fait en deux parties :

— à l'initialisation, `lo = 0` et `hi = n`, donc les propriétés sont triviales,
— à chaque itération, si elles sont vraies au début, elles restent vraies après mise à jour de `lo` ou `hi`.

On en déduit que ces invariants sont toujours vrais à la fin de l'exécution, et que v ne se trouve dans aucun intervalle éliminé. Comme ces intervalles couvrent tout le tableau, on en conclut que a ne contient aucune occurrence de v.

Sûreté.

On a pu justifier que tout résultat renvoyé par `binary_search` est correct. Mais cette fonction renvoie-t-elle effectivement toujours un résultat ?

Deux éléments présentent un risque :

— l'accès à `a[mid]` suppose que `mid` est un indice valide,
— la boucle `while (lo < hi)` doit finir par s'arrêter.

Pour montrer que la fonction est sûre, on vérifie :

— que $0 \leq lo \leq hi \leq n$ est toujours vrai (invariant),
— que `mid` vérifie toujours $lo \leq \texttt{mid} < hi$.

Terminaison.

Pour montrer que la fonction termine, on remarque que la longueur de l'intervalle $hi - lo$ diminue strictement à chaque étape. Cette quantité est un *variant*. Elle ne peut pas décroître indéfiniment, donc la boucle s'arrête nécessairement.

Performances.

Le temps d'exécution d'un programme découle directement des différentes opérations réalisées, et du nombre de fois où chacune est réalisée. Dans `binary_search`, le détail des opérations varie selon l'issue des différents tests. On peut répertorier :

— l'initialisation des deux variables `lo` et `hi`,
— le test `lo < hi` évalué avant chaque tour de boucle, plus éventuellement une fois à l'achèvement de la boucle,
— à chaque exécution du corps de la boucle, un maximum de dix opérations variées : opérations arithmétiques, affectations, tests ou accès aux cases du tableau.

En ajoutant tout cela on dénombre un maximum de $2 + (N + 1) + 10N = 3 + 11N$ opérations, où N est le nombre de tours de boucle effectués. La valeur précise de cette somme n'est pas forcément significative : les différentes opérations additionnées ne sont pas toutes comparables. On retient en revanche une information clé : la quantité totale d'opérations est proportionnelle au nombre de tours effectué par la boucle `while`.

Le point déterminant dans l'évaluation de la complexité de notre fonction `binary_search` est donc l'estimation du nombre de tours de boucle. Pour cela, rappelons qu'à chaque étape on réduit de moitié la taille de l'intervalle de recherche : le nombre maximum de tours de boucle est donc le nombre de fois que l'on peut diviser la taille du tableau par deux avant d'atteindre zéro. Plus précisément, on peut montrer que si à une étape donnée, `hi - lo` $< 2^k$, alors il reste k tours de boucle au maximum avant l'arrêt du programme. On en déduit que `binary_search` cherche un élément dans un tableau a de taille n avec un nombre total d'opérations proportionnel au logarithme de n.

Si l'on considère la solution plus simple `sequential_search`, on a toujours un nombre total d'opérations proportionnel au nombre de tours de boucle, mais avec un nombre de tours cette fois susceptible d'être égal à la taille n du tableau. Cette version est donc considérablement plus coûteuse pour les grands tableaux, et l'écart de performances observé empiriquement était donc tout à fait prévisible.

Bilan.

En observant le code d'un programme, on peut en apprendre beaucoup sur son comportement. Une telle étude permet notamment de raisonner sur les programmes et les algorithmes, analyser mathématiquement leurs performances et démontrer leur bon fonctionnement. Dans ce chapitre, nous allons développer différentes techniques pour cette étude formelle et rigoureuse des programmes et algorithmes, qui vient compléter les tests et les évaluations expérimentales que nous savions déjà réaliser.