**ALGORITHMES DE RECHERCHE**

Sommaire

[Introduction 2](#_Toc9865600)

[Algorithmes de recherches 3](#_Toc9865601)

[Recherche séquentielle 3](#_Toc9865602)

[Définition 3](#_Toc9865603)

[Algorithme 3](#_Toc9865604)

[Code C 3](#_Toc9865605)

[Recherche dichotomique 3](#_Toc9865606)

[Définition 3](#_Toc9865607)

[Algorithme 4](#_Toc9865608)

[Code C 4](#_Toc9865609)

[Recherche ternaire 5](#_Toc9865610)

[Définition 5](#_Toc9865611)

[Algorithme 5](#_Toc9865612)

[Code C 6](#_Toc9865613)

[Recherche sautée 7](#_Toc9865614)

[Définition 7](#_Toc9865615)

[Algorithme 7](#_Toc9865616)

[Code C 8](#_Toc9865617)

[Recherche par interpolation 8](#_Toc9865618)

[Définition 8](#_Toc9865619)

[Algorithme 9](#_Toc9865620)

[Code C 10](#_Toc9865621)

[Recherche exponentielle 10](#_Toc9865622)

[Définition 10](#_Toc9865623)

[Algorithme 11](#_Toc9865624)

[Code C 11](#_Toc9865625)

[Recherche de Fibonacci 11](#_Toc9865626)

[Définition 11](#_Toc9865627)

[Algorithme 12](#_Toc9865628)

[Code C 13](#_Toc9865629)

[Utilisation dans le jeu vidéo 15](#_Toc9865630)

[Conclusion 15](#_Toc9865631)

[Crédits 16](#_Toc9865632)

[Codes complets 16](#_Toc9865633)

# **Introduction**

L'algorithme de recherche est un type spécifique d'algorithme. Il permet de retourner un résultat à partir d'un problème donné. En somme, comme son nom l'indique, il retourne une position pour donner suite à une recherche de valeur dans un tableau. Il y a deux grands types d'algorithme de recherches : la recherche séquentielle, et la recherche dichotomique.

La recherche séquentielle, aussi appelée recherche par balayage consiste à parcourir un tableau à partir du premier élément, et à s'arrêter lorsque l'on a trouvé. Le temps de recherche est séquentiel, puisque l'on peut parcourir tous les éléments du tableau. Lorsque la valeur est atteinte, on renvoie son indice ; sinon, on continue. Celui-ci ne nécessite pas d'avoir des données triées. Cependant, ce n'est pas toujours la solution la plus efficace pour la recherche de données.

La recherche dichotomique, elle, concerne une recherche dans un tableau déjà trié. "Diviser pour mieux régner" est la devise de cette recherche. Elle fonctionne par comparaison entre les éléments du tableau, et est plus rapide que la recherche séquentielle. Le principe est, contrairement à la recherche séquentielle, exponentiel. On commence la recherche au milieu du tableau, en trouvant la position la plus centrale. Si le tableau est vide, on sort. Ensuite, on compare la valeur recherchée avec celle médiane. Si ces deux valeurs sont égales, on renvoie la position. Sinon, on continue la recherche dans la partie pertinente du tableau.

Ici, nous allons ainsi vous présenter plusieurs algorithmes de recherche spécifiques.

# **Algorithmes de recherches**

## Recherche séquentielle

### Définition

Aussi appelé recherche linéaire. La recherche séquentielle est la recherche la plus simple et la plus basique. Le principe est de défiler dans le tableau de données et de les vérifier les unes après les autres jusqu’à trouver la donnée voulue. Cette recherche n’est pas la plus efficace ni la plus rapide.

### Algorithme

FONCTION rechercheSéquentielle (tab:TABLEAU DE ENTIER; x,n : ENTIER)

VAR

i,res: ENTIER

DEBUT

res <- -1

POUR i DE 1 A n FAIRE

SI tab(i)=x ALORS

res <- i

FINSI

FINPOUR

rechercheSéquentielle <- res

FINFONCTION

### Code C

int rechercheSequentielle(int tab[], int n, int x)

{

int i;

for (i = 0; i < n; i++)

if (tab[i] == x)

return i;

return -1;

}

## Recherche dichotomique

### Définition

Aussi appelé recherche binaire. La recherche dichotomique est basée sur la division de la chaine en deux parties. Le principe est de diviser le tableau en deux parties égales puis de regarder dans quelle partie se situe la donnée recherchée. On répète la même étape jusqu’à isoler la donnée recherchée. Cette recherche reste longue et peu efficace.

### Algorithme

FONCTION rechercheDichotomique (tab:TABLEAU DE ENTIER; l, r, x:ENTIER) DE ENTIER

VAR

mid,res : ENTIER

DEBUT

// Si l’element recherché est au milieu, alors on renvoie mid

SI tab(mid) = x ALORS

res <- mid

SINON

// Si l’element est plus petit que mid, alors

// il est present dans la partie de gauche

SI tab(mid) > x ALORS

res <- rechercheDichotomique(tab, l, mid-1, x)

SINON

// Sinon l’element est present

// dans la partie de droite

SI tab(mid) < x ALORS

res <- rechercheDichotomique(tab, mid+1, r, x)

SINON

res <- -1

FINSI

FINSI

FINSI

FINFONCTION

### Code C

int rechercheDichotomique(int tab[], int l, int r, int x) {

if (r >= l) {

int mid = l + (r - l) / 2;

// Si leelement recherche est au milieu, alors on renvoie mid

if (tab[mid] == x)

return mid;

// Si leelement est plus petit que mid, alors

// il est present dans la partie de gauche

if (tab[mid] > x)

return rechercheDichotomique(tab, l, mid - 1, x);

// Sinon l'element est present

// dans la partie de droite

return rechercheDichotomique(tab, mid + 1, r, x);

}

return -1;

}

## Recherche ternaire

### Définition

La recherche tertiaire est assez similaire à la recherche dichotomique. Le principe est de séparer le tableau en trois parties et de regarder dans quelle partie se situe la donnée recherchée. La même étape est ensuite répétée jusqu’à ce que la donnée soit isolée.

### Algorithme

FONCTION recherche\_Tertiaire(mid1,mid2,r,key : ENTIER) DE ENTIER

VAR

res : ENTIER

DEBUT

res <- -1

SI(r>=1) ALORS

mid1=1+(r-1)/3

mid2=r(r-1)/3

FINSI

SI(mid1=key) ALORS

res <- mid1

FINSI

SI(mid2=key) ALORS

res <- mid2

FINSI

SI(key<mid1) ALORS

res<-recherche\_Tertiaire() // On relance la fonction

FINSI

SI(key>mid2) ALORS

res<-recherche\_Tertiaire() // On relance la fonction

SINON

res<-recherche\_Tertiaire() // On relance la fonction

FINSI

recherche\_Tertiaire <- res

FINFONCTION

### Code C

int rechercheTernaire(int l, int r, int key, int tab[]) {

if (r >= l) {

// On cherche mid1 and mid2

int mid1 = l + (r - l) / 3;

int mid2 = r - (r - l) / 3;

// On vérifie si l’element est présent à l’un des mid

if (tab[mid1] == key) {

return mid1;

}

if (tab[mid2] == key) {

return mid2;

}

// Sinon si L'objet n’est pas présente à l’un des deux mid,

// On vérifie dans quelle partie l’element est present

// Puis on répète l’opération

// dans la partie ciblée

if (key < tab[mid1]) {

// L'objet se trouve entre l et mid1

return rechercheTernaire(l, mid1 - 1, key, tab);

}

else if (key > tab[mid2]) {

// L'objet se trouve entre mid2 et r

return rechercheTernaire(mid2 + 1, r, key, tab);

}

else {

// L'objet se trouve entre mid1 et mid2

return rechercheTernaire(mid1 + 1, mid2 - 1, key, tab);

}

}

// L'objet n’a pas été trouvé

return -1;

}

## Recherche sautée

### Définition

Comme pour la recherche dichotomique et ternaire, la recherche sautée est utilisée dans des tableaux triés. Elle créée un bloque et essaye de trouver l'objet recherché dans ce bloc, s'il n'est pas dans le bloc, l'algorithme le déplace et cherche dans ce nouveau bloc jusqu'à que l'objet est trouvé ou que l'algorithme est arrivé à la fin du tableau.

La taille du bloc est basée sur la taille tu tableau, s'il le tableau est de taille *n* alors le bloc est de taille *√n*.

Après avoir trouvé le bloc où se trouve l'objet, l'algorithme fait une recherche séquentielle dans le bloc.

La performance de l'algorithme de recherche sautée se trouve entre l'algorithme de recherche séquentielle et dichotomique.

### Algorithme

FONCTION rechercheSautee (nTab:TABLEAU DE ENTIER; nTaille,nObjet:ENTIER) DE

ENTIER

VAR

nI, nDebut, nFin, res : ENTIER

DEBUT

res <- -1

nI <- 0

nDebut <- 0

nFin <- nTaille/nTaille //La taille du bloc est la racine carrée de la longueur du tableau

//On verifie si on a le bon bloc

TANTQUE (nTab(nFIN) <= nObjet ET nFin < nTaille) FAIRE

nDebut <- nFin;

nFin <- nFIn + nTaille/nTaille

SI nFin > nTaille -1 ALORS

nFIn <- nTaille //Si la fin du bloc dépasse la taille du tableau alors on la corrige

FINSI

FINTANTQUE

//Recherche séquentielle dans le bloc séléctionné

POUR nI DE nDebut A nFin FAIRE

SI nTab(nI) = nObjet ALORS

res <- nI

FINSI

FINPOUR

rechercheSautee <- res

FINFONCTION

### Code C

int rechercheSautee(int nTab[], int nTaille, int key) {

int nI = 0;

int nDebut = 0;

int nFin = sqrt(nTaille); //La taille du bloc est la racine quarre de la longueur du tableau

//On verifie si on a le bon bloc

while(nTab[nFin] <= key && nFin < nTaille) {

nDebut = nFin; //ce n'est pas le bon bloc, alors on le deplace

nFin += sqrt(nTaille);

if(nFin > nTaille - 1){

nFin = nTaille; //Si la fin du bloc depasse la taille du tableau alors on la corrige

}

}

//Recherche sequentielle dans le bloc selectionne

for(int nI = nDebut; nI < nFin; nI++) {

if(nTab[nI] == key){

return nI; //On retourne l'index de l'objet trouve

}

}

return -1;

}

## Recherche par interpolation

### Définition

Le principe est similaire à celui de la recherche dichotomique : il s'agit de comparer un élément du tableau avec la valeur recherchée, puis de rechercher récursivement dans la portion du tableau pertinente. La différence est sur le choix de la valeur du tableau choisit.

Dans une recherche dichotomique c'est la médiane qui est utilisée. Ce choix n'est pas optimal si l'on sait que les valeurs sont "bien reparties", par exemple pour rechercher le mot "banane" dans un dictionnaire, il est plus pertinent de chercher au début du dictionnaire. Ainsi dans la recherche par interpolation, c'est la position qui correspond à la place de l'élément cherché, si les données étaient régulièrement espacées entre le minimum et le maximum de la liste, qui est étudiée.

### Algorithme

FONCTION rechercheInterpolation(tab:TABLEAU DE ENTIER, n,x: ENTIER) DE ENTIER

VAR

debut,fin,res,pos : ENTIER

DEBUT

//initialisatioin des variables

debut <- 0

fin <- n-1

TANTQUE (debut <= fin ET x >= tab(debut) ET x <= tab(fin) OU res = -1) ALORS

SI debut = fin ALORS

SI tab(debut) = x ALORS

res <- debut

SINON

res <- -1

FINSI

FINSI

pos <- debut + (fin-debut) / (tab(fin)\_tab(debut))\*(x-tab(debut))

SI tab(pos) = x ALORS

res <- pos

FINSI

// Si l'objet est suppérieur, alors il est dans la partie suppérieure

SI tab(pos) < x ALORS

debut <- pos + 1

SINON

// Si l'objet est suppérieur, alors il est dans la partie suppérieure

fin <- pos -1

FINSI

FINTANTQUE

rechercheInterpolation <- res

FINFONCTION

### Code C

int interpolationSearch(int tab[], int n, int x) {

//Initialisation des index de début et fin

int debut = 0, fin = (n - 1);

while (debut <= fin && x >= tab[debut] && x <= tab[fin])

{

if (debut == fin){

if (tab[debut] == x) return debut;

return -1;

}

//Recherche de la position de l'objet recherché avec une formule d'interpolation

int pos = debut + (((double)(fin-debut) / (tab[fin]-tab[debut]))\*(x - tab[debut]));

//Si l'objet recherché est trouvé

if (tab[pos] == x)

return pos;

//Si l'objet est suppérieur, alors il est danss la partie suppérieure

if (tab[pos] < x)

if (tab[pos] < x)

debut = pos + 1;

// Si l'objet est inférieur, alors il est danss la partie inférieure

else

fin = pos - 1;

}

return -1;

}

## Recherche exponentielle

### Définition

La recherche exponentielle (aussi appelée recherche doublante ou galopante) est un mécanisme utilisé pour trouver le rang où la clé de recherche est présente. Si L et U sont les liens en haut et en bas de la liste, alors L et U sont tous les deux puissances de 2. Pour la dernière section, le U est la dernière position de la liste. C’est pour cette raison qu’on l’appelle exponentielle.

Après avoir trouvé le rang spécifique, on utilise la recherche dichotomique pour trouver la position exacte de la clé de recherche.

Un tableau trié et un élément x qui doit être recherché, on trouve la position de x dans le tableau.

### Algorithme

FUNCTION rechercheExponentielle(tableau() : TABLEAU DE ENTIER; n,x : ENTIER)

VAR

res,i : ENTIER

DEBUT

SI tableau(o) = x) ALORS

res <- 0

FINSI

i <- 1

TANTQUE (i < n ET tableau(i) <= x) FAIRE

i <- i\*2

FINTANTQUE

rechercheDichotomique(tableau, i/2, min(i,n), x) //chercher l’objet dans l’intervalle réduit avec une recherche dichotomique

FINFONCTION

### Code C

int rechercheExponentielle(int tableau[], int n, int x){

if (tableau[0] == x)

return (0);

int i = 1;

while (i < n && tableau[i] <= x)

i = i\*2;

return (rechercheBinaire(tableau, i / 2, min(i, n), x)); //chercher l’objet dans l’intervalle réduit avec une recherche dichotomique

}

## Recherche de Fibonacci

/!\ Attention le code les l'algorithme de Julien de fonctionnent pas. /!\

/!\Demandez lui de finir son travail/!\

### Définition

La recherche de Fibonacci fonctionne de la manière suivante : en premier on trouve le plus petit chiffre de Fibonacci qui est supérieur ou égal à la taille de notre liste, soit *fib* le nombre trouvé. On utilise *fib*-2 comme index (s'il est valide). Soit *i* le nombre *fib*-2, on compare arr[*i*] avec x (objet recherché), s'ils sont les mêmes, on retourne *i*. Si si x est supérieur, on relance pour sous-tableau après *i*, sinon on relance pour sous-tableau avant *i*.

### Algorithme

FONCTION fibMonaccianSearch(arr[] : TABLEAU D'ENTIER, x : ENTIER, n : ENTIER)

VAR

fibMMm2 <- 0 : ENTIER

fibMMm1 <- 1 : ENTIER

fibM <- fibMMm2 + fibMMm1

DEBUT

TANT QUE (fibM < n)

fibMMm2 <- fibMMm1

fibMMm1 <- fibM

fibM <- fibMMm2 + fibMMm1

FINTANTQUE

offset <- -1 : ENTIER

//alors qu'il y a des éléments à inspecter. Nous comparons arr [fibMm2] avec x. Quand fibM devient 1, fibMm2 devient 0\*/

TANT QUE (fibM > 1)

// Vérifiez si fibMm2 est un emplacement valide

VAR i <- min(offset+fibMMm2, n-1)

// Si x est supérieur à la valeur d'indice fibMm2,couper le tableau de sous-tableau de l'offset à i

SI (arr[i] < x)

DEBUT

fibM <- fibMMm1

fibMMm1 <- fibMMm2

fibMMm2 <- fibM - fibMMm1

offset <- i

FIN

// Si x est supérieur à la valeur d'indice fibMm2, couper le sous-tableau après i + 1

SINON SI (arr[i] > x)

DEBUT

fibM <- fibMMm2

fibMMm1 <- fibMMm1 - fibMMm2

fibMMm2 <- fibM - fibMMm1

FIN

// élément trouvé. indice de retour

SINON fibMonaccianSearch <- i

FINSI

FINSI

FINTANTQUE

// comparer le dernier élément avec x

SI(fibMMm1 ET arr[offset+1] = x)

DEBUT

fibMonaccianSearch <- offset+1

FIN

// élément non trouvé. retour -1\*/

fibMonaccianSearch <- -1

FIN FONCTION

### Code C

/\* Retourne l’index de x SI existe, SINON retourne -1 \*/

int fibMonaccianSearch(int arr[], int x, int n)

{

/\* Initialisation des nombres Fibonacci \*/

int fibMMm2 = 0; // (m-2)'th Fibonacci No.

int fibMMm1 = 1; // (m-1)'th Fibonacci No.

int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci

/\* fibM va stocké le plus petit nombre de Fibonacci

     Plus grand ou égal à n\*/

while (fibM < n)

{

fibMMm2 = fibMMm1;

fibMMm1 = fibM;

fibM = fibMMm2 + fibMMm1;

}

// Marque les valeurs éliminées depuis le début

int offset = -1;

/\* alors qu'il y a des éléments à inspecter. Nous comparons arr [fibMm2] avec x. Quand fibM devient 1, fibMm2 devient 0\*/

while (fibM > 1)

{

// Vérifiez si fibMm2 est un emplacement valide

int i = min(offset+fibMMm2, n-1);

/\* Si x est supérieur à la valeur d'indice fibMm2,

couper le tableau de sous-tableau de l'offset à i \*/

if (arr[i] < x)

{

fibM = fibMMm1;

fibMMm1 = fibMMm2;

fibMMm2 = fibM - fibMMm1;

offset = i;

}

/\* Si x est supérieur à la valeur d'indice fibMm2,

couper le sous-tableau après i + 1\*/

else if (arr[i] > x)

{

fibM = fibMMm2;

fibMMm1 = fibMMm1 - fibMMm2;

fibMMm2 = fibM - fibMMm1;

}

/\* élément trouvé. indice de retour \*/

else return i;

}

/\* comparer le dernier élément avec x \*/

if(fibMMm1 && arr[offset+1]==x)return offset+1;

/\* élément non trouvé. retour -1\*/

return -1;

}

# **Utilisation dans le jeu vidéo**

Les algorithmes de recherches sont aussi utilisés dans le domaine du jeu vidéo tels que :

* L’Intelligence Artificielle :

L'intelligence Artificielle est souvent utilisée pour la création des bots dans le milieu des jeux vidéo, qui sont conçus pour servir d'opposants.

L'intelligence artificielle dispose des mêmes contrôles que le joueur. Celle-ci peut aussi permettre au joueur d'apprendre plus facilement.

* Le Pathfinding :

Le pathfinding est un problème se rattachant au domaine de la planification et à la recherche de solution telles que les personnages de jeux vidéo, se déplacent dans un environnement en évolution réelle.

* Les jeux combinatoires (algorithme minmax) :

Les jeux combinatoires sont nés en 1900 et sont devenus des sujets de recherche à part entière dans les domaines scientifiques et mathématiques. Les jeux combinatoires sont des jeux à deux joueurs (ou bien contre une IA) disposant de placements précis se jouant au tour par tour en suivant des règles données en essayant d’obtenir la victoire. Les jeux combinatoires ne disposent pas de fins aléatoires.

* Terraforming :

Le terraforming représente la formation de terrain. Celle-ci peut être aléatoire (procédurale comme Minecraft) ou bien générée par le joueur ou une IA.

# **Conclusion**

Nous avons donc abordé les différentes méthodes de recherches et avons pu mettre en lumière les spécificités de chacune d’entre elles.

Si les différences entre les méthodes semblent parfois relativement restreintes, on constate qu’elles peuvent cependant mener à des utilisations très différentes et variées. Ce large panel de possibilités permet de régler de nombreuses problématiques présentes dans le développement de jeu vidéo (entre autres) et simplifie grandement la tâche, nous avons ainsi listé les principaux aspects concernés.

**Crédits**

* Lucas Aulen
  + Documentation finale
  + Recherche sautée
  + Recherche interpolée
  + Corrections et ajustements
* Yoann Fernandes
  + Recherche séquentielle
  + Recherche Dichotomique
  + Recherche Ternaire
* Pablo de Chénerilles
  + Recherche exponentielle
  + Conclusion
* Julien Cragnolini
  + Recherche de Fibonacci
* Justine Buttani
  + Introduction
* Mathieu Haas
  + Utilisation dans le jeu-vidéo

# Codes complets

|  |  |
| --- | --- |
| Recherche séquentielle |  |
| Recherche dichotomique |  |
| Recherche ternaire |  |
| Recherche sautée |  |
| Recherche par interpolation |  |
| Recherche exponentielle |  |
| Recherche de Fibonacci | Pas de code fonctionnel |