Intro complexité

9 septembre 2024 16:33

- Combien de temps prend find?

Dépend de: la machine ✗ nb instr
de n = nbelem ✓
de si trouvé tôt ou pas ✗ pire cas

- On veut estimer le temps requis en fct de n seulement.
Temps =
nb d'instructions exécutées
en fct de n
dans le pire cas

C'est quoi une instruction?
$\approx$ ligne de code

ex: find exécute $\approx 5 \times n + 1$ instructions
ou $6n+1$

En tout cas, $\approx$ "une valeur indépendante de n" $\times n + 1$

Autrement dit, $\approx c \times n + 1$ où c est une constante

Aussi, le $+1$ est une constante, disons d

Donc temps $c \times n + d$
Bref, une fonction linéaire $\Rightarrow O(n)$

```
void f(){
  for(int i=0;i<n; ++i){
    for (int j=0; j<n; ++i){
      cout<< tab[i]+tab[j];
    }
  }
}
```

bloc exécuté ≈ n fois

n fois × n fois → n^2 fois

Temps de f: $\approx c \times n^2 + dn + e$

Bref, une fonction quadratique $\Rightarrow O(n^2)$

- La notation O sert à isoler le <u>terme dominant</u> d'une fonction (ici, le nb d'instructions).

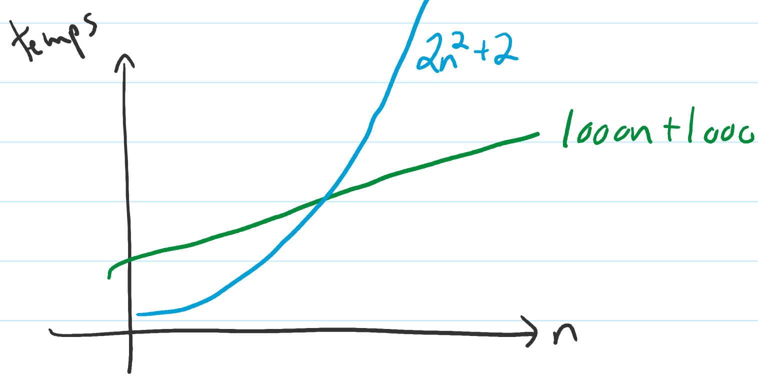
  Elle fait abstraction des constantes et ressort <u>l'ordre de grandeur</u> de la fonction.

- Idée: quand n devient grand, par ex. si on stocke $n=1000000$ élts, les constantes ont peu d'impact comparé à l'ordre de grandeur.

- ex: deux algos de find

	nb instructions	n=10	n=1000	n=10000
1)	$1000 \times n + 1000$	11000	1,001,000	10,001,000
2)	$2n^2+2$	202	2000002	200000002

2) $2n^2+2$ | 202 2,000,002 200,000,002



- En iFT339, n = nb d'elts stockés
- $O(n)$ veut dire "à une constante près de n"

$O(f(n))$

- iFT436 = def précise
- Ici, on applique O à une qté de temps, mais O peut mesurer n'importe quoi.

ex: $O(n)$ espace mémoire
$O(n^2)$ espace mémoire (votre matrice)
etc.

- Pour obtenir la complexité O en temps d'une fct

1) Calculer le nb d'instructions — ex $4n^2+2n-4$

2) Garder le terme (positif) dominant — $4n^2$

2) Garder le terme (positif) dominant $4n^2$

$n^3 > n^2 > n > \log n > \log(\log n) \ldots$

3) Enlever la constante n^2
└ donne le terme O $O(n^2)$

ex: $50n^2 + \underline{0.1n^3} - n + 4$

$0.1n^3$

$O(n^3)$

ex: $3\log n + \frac{1}{2}$

$\log n$

$O(\log n)$

- Ex: vector::reserve → une boucle itère de 0 à n=nbelem
  → $c \cdot n$
  → $O(n)$

- Ex: vector::push_back: dans le <u>pire cas</u>, appelle reserve
  → $O(n)$

- Ex : cours O

```
vector<int> v;
//...

for(int i=0; i< v.size(); ++i){
    v.find(i);   ] temps ≈ n        ] n fois
}
```

n fois une temps $\approx c \cdot n \rightarrow n \cdot n \cdot c \rightarrow O(n^2)$

ex : vector::at : 2 instructions

$2 \rightarrow O(?)$

Si la fonction ne dépend aucunement de n, on écrit $O(1)$

$O(1)$ = temps constant = hyper-rapide !!

Ordres de grandeur

$O(1)$ → temps constant, indép. de n

→ $O(\log n)$ → très rapide

$O(n)$ → simple(s) boucle(s)   for(i=0..n-1) ...

$O(n^2)$ → boucles imbriquées   for(i=0..n-1)

$O(n) \rightarrow$ simple(s) boucle(s)

```
for(i=0..n-1)
  ...
```

$O(n^2) \rightarrow$ boucles imbriquées

```
for(i=0..n-1)
  for(j=0..n-1)
    ...
```

$O(n^3) \rightarrow$

```
for
  for
    for
```

$\vdots$

$O(2^n) \rightarrow$ hyper-lent

$O(n!) \rightarrow O(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1)$

Fouille binaire / recherche dichotomique

↳ si tab est un tableau trié (ou vecteur ou autre)

on peut faire find en temps $O(\log n)$

tab: 2 4 9 12 14 21 30

9<12, donc chercher à gauche

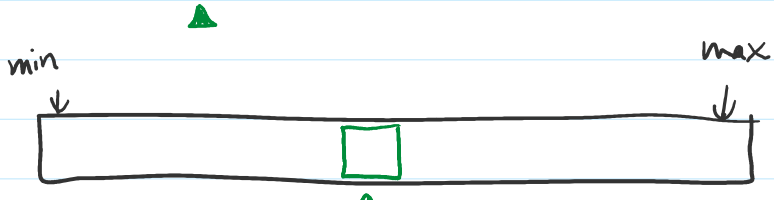
find(9)

2 4 9

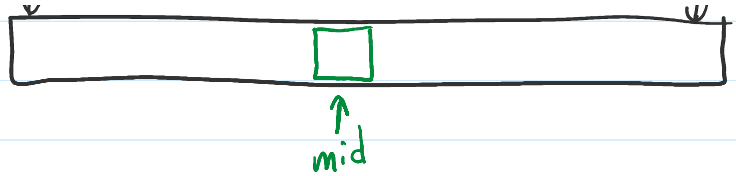
9>4 → droite

9 trouvé

find(x) tab:

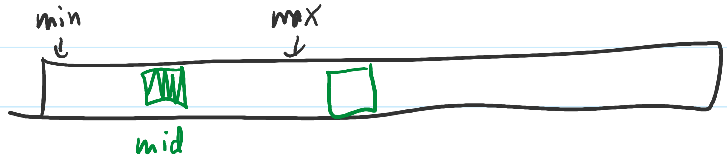


find(x) tab:

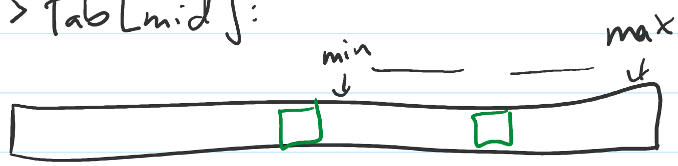


si tab[mid]==x, return mid

si x < tab[mid]:



si x > tab[mid]:




```
int fouilleDicho( tab, n, x){
    int min=0, max = n-1;

    while ( min <= max ){
        int mid = (max-min)/2 +min;
        if ( tab[mid] < x )
            min =mid+1;
        else if (tab[min] > x)
            max = mid-1;
        else
            return mid;
    }
    return -1;
}
```

Combien de temps? Chaque itération prend un temps $O(1)$,
donc temps = $c \times$ nbiter

Combien de temps ; Chaque itération prend un temps $O(1)$,
donc temps = $c \times$ nbiter

Chaque iter réduit l'espace de recherche d'une moitié

iter	nb elem à considérer		
0	n		
1	$n/2$	$\div 2$	
2	$n/4$	$\div 2$	
3	$(n/4)/2 = n/8$	$\div 2$	
$\vdots$		$\vdots$	
k	1	$\div 2$	k = nbiter

k = Nb iter = nb de fois qu'il faut diviser n par 2 pour atteindre 1.

$$1 = n \cdot \underbrace{\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \ldots \cdot \frac{1}{2}}_{k \text{ fois}}$$

$$1 = n \cdot \frac{1}{2}^k \rightarrow 2^k = n$$

$$\log_2(2^k) = \log_2(n)$$

$$k \cdot \log_2(2) = \log_2(n)$$

$$k = \log_2(n)$$

• Complexité à 2 paramètres

```
Matrice m(n,m);

for (int i=0; i<n; ++i){
    for (int j=0; j<m; ++j) {        ~n fois

        m(i,j)=0;        n·m fois
    }
}
```

Temps? $a \cdot n + b \cdot n \cdot m$ fois a,b constantes

$O(nm)$

ex: $2n^2 + 10nm$

$O(n^2)$? Sous estime si $m = n^{10}$
$O(nm)$? Sous estime si $n = m^{10}$

$O(n^2 + nm)$

• Grand-O d'une fct à 2 params n et m

1) Conserver <u>les</u> termes dominants
   └ si on ne sait pas qui domine qui entre 2 termes, garder les 2

2) enlever les constantes multiplicatives

2) enlever les constantes multiplicatives

ex: $\underbrace{0.1\,n^2}_{\text{dominé par } 10n^3m} + 10n^3m + \underbrace{\log m \cdot n^2}_{n^2 \log m \text{ dominé par } 10n^3 m \text{ car } n^2 < n^3,\ \log m < m} + m^2 \log n$

$$10n^3m + m^2 \log n$$

$$O(n^3m + m^2 \log n)$$

$f(n,m)$ est dominé par $g(n,m)$ s'il existe une constante c telle que

$$f(n,m) \leq c \cdot g(n,m)$$

- Complexité de $n \times$ push_back

```
vector<int> v;
for (i=0..n)
    v.push_back(i);
```

Analyse naïve: push_back prend $O(n)$
On le fait n fois
$\rightarrow O(n \cdot n) = O(n^2)$

Analyse poussée: On suppose que $n = 2^k + 1$
Push sans copie: $O(1)$

Push sans copie : $O(1)$

avec copie : $O(nbelem)$ au moment du push

	i = nbelem	cap	Temps de copie	Temps pour ajouter i
	1	1	0	1
2^0+1	2	$1 \to 2$	1	1
2^1+1	3	$2 \to 4$	2	1
	4	4	0	1
2^2+1	5	$4 \to 8$	4	1
	6	8	0	1
	7	8	0	1
	8	8	0	1
2^3+1	9	$8 \to 16$	8	1
	10		0	1
	$\vdots$		$\vdots$	
	16		0	1
	17	$16 \to 32$	16	1
		$\vdots$		1
	$n = 2^k$	2^k	0	1
	$n = 2^k+1$	$2^k \to 2 \cdot 2^k$	2^k	1
			total = ?	total : n

Copie : pour chaque 2^j+1, on paie 2^j

$$\Rightarrow \text{Total} = 2^0 + 2^1 + 2^2 + \dots + 2^k$$

$$= \sum_{i=0}^{k} 2^i = 2 \cdot 2^k - 1 \quad \text{(MAT115)}$$

$$n = 2^k + 1 \to 2^k = n-1 \;\Big|\; 2 \cdot (n-1) - 1 \to 2n - 3$$

$n = 2^k + 1 \rightarrow 2^k = n - 1 \mid 2 \cdot (n-1) - 1 \rightarrow 2n - 3$

$O(n)$

- Si on augmentait cap+1 au lieu 2*cap, chaque i ferait i-1 copies

↳ $\sum_{i=1}^{n} (i-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \rightarrow O(n^2)$