Algorithmique pour l'Algèbre Linéaire

Thomas Fernique
CNRS & Univ. Paris 13

L1 Maths-Info
24 mars 2017

Menu

Menu

Tableaux en C

Que fait la fonction suivante ?

```
float * blup(float* P, int p, float* Q, int q)
{
 int i,j;
 float* M = malloc((p+q)^2,sizeof(float));
 for (i=0; i<q; i++)
  for (j=0; j<=p; j++)
   M[i*(p+q)+i+j]=P[p-j];
 for (i=0; i<=q; i++)
  for (j=0; j<p; j++)
   M[(j+q)*(p+q)+i+j]=Q[q-i];
 return M;
}
```

Complexité

La *complexité* d'un algorithme est le nombre $c(n)$ d'opérations effectuées pour traiter des données de taille n.

On s'intéresse au comportement *asymptotique* (n grand).
On utilise la notation O pour simplifier l'analyse :

$$5n^2 + 123n - \log(n) = O(n^2),$$

où $f = O(g)$ quand f/g est uniformément bornée.

Menu

Principe

Classe d'algorithmes qui marchent en trois temps :

- **Diviser** : découper le problème initial en sous-problèmes ;
- **Régner** : résoudre les sous-problèmes récursivement ;
- **Combiner** : en déduire une solution au problème initial.

Recherche dichotomique

Bob doit deviner un nombre entre 1 et n choisi par Alice.
Celle-ci lui dit “plus grand” ou “plus petit” à chaque essai raté.

Plutôt que d’essayer les n nombres, Bob peut diviser pour régner :

- Diviser : Bob propose le nombre médian ;
- Régner : il relance récursivement sur le bon intervalle ;
- Combiner : Bob a juste à se rappeler du nombre trouvé.

Complexité

Le nombre maximal $C(n)$ de propositions de Bob vérifie

$$C(n) = C\left(\frac{n}{2}\right) + 1 = C\left(\frac{n}{2^2}\right) + 2 = \ldots = C\left(\frac{n}{2^p}\right) + p.$$

Avec $p = log_2(n)$ on a $2^p = n$ et donc $C(n) = O(\log_2(n))$.

L'algorithme est linéaire en la taille des données (le nombre n).

Produit de deux entiers (1/2)

Algo. naïf : multiplier chaque chiffre de x avec chaque chiffre de y.

$$79 \times 81 = (7 \times 8).10^2 + (7 \times 1 + 9 \times 8).10 + 9 \times 1.$$

Produit de deux entiers (1/2)

Algo. naïf : multiplier chaque chiffre de x avec chaque chiffre de y.

$$79 \times 81 = (7 \times 8).10^2 + (7 \times 1 + 9 \times 8).10 + 9 \times 1.$$

Dans les années 60, Anatolii Karatsuba remarque que

$$7 \times 1 + 9 \times 8 = (7 + 9) \times (1 + 8) - 7 \times 8 - 9 \times 1,$$

soit trois multiplications au lieu de deux . . . mais deux déjà faites !

Produit de deux entiers (1/2)

Algo. naïf : multiplier chaque chiffre de x avec chaque chiffre de y.

$$79 \times 81 = (7 \times 8).10^2 + (7 \times 1 + 9 \times 8).10 + 9 \times 1.$$

Dans les années 60, Anatolii Karatsuba remarque que

$$7 \times 1 + 9 \times 8 = (7 + 9) \times (1 + 8) - 7 \times 8 - 9 \times 1,$$

soit trois multiplications au lieu de deux . . . mais deux déjà faites !

Quel rapport avec diviser pour régner ?

Produit de deux entiers (2/2)

La remarque de Karatsuba permet de faire moins de produits :

Diviser : écrire $x = a.10^n + b$ et $y = c.10^n + d$;

Régner : calculer ac, bd et $(a + b)(c + d)$ récursivement ;

Combiner : faire 3 multiplications au lieu de 4 via la formule :

$$(a.10^n+b)(c.10^n+d) = ac.10^{2n}+[(a+b)(c+d)-ac-bd].10^n+bd.$$

En C

```
int karatsuba(int x, int y, int nbits)
{
 if (nbits<=3) // sinon nbits-m+1=nbits -> sans fin
  return x*y;
 int m=nbits/2;
 int a=x>>m;
 int b=x-(a<<m);
 int c=y>>m;
 int d=y-(c<<m);
 int ac=karatsuba(a,c,nbits-m);
 int bd=karatsuba(b,d,m);
 int abcd=karatsuba(a+b,c+d,nbits-m+1);
 return ((ac<<(2*m))+((abcd-ac-bd)<<m)+bd);
}
```

Complexité

On la mesure ici par le nombre de produits de deux chiffres.
$C(n)$: complexité pour multiplier deux entiers à n chiffres.

Algo. naïf : $C(n) = n^2$ (quadratique en la taille des données).

Algo. de Karatsuba :

$$C(n) = 3C\left(\frac{n}{2}\right) = \ldots = 3^p C\left(\frac{n}{2^p}\right) = O(3^{\log_2(n)})$$

On y gagne car

$$3^{\log_2(n)} = 2^{\log_2(n)\log_2(3)} = n^{\log_2(3)} \quad \text{et} \quad \log_2(3) \simeq 1,58\ldots$$

Menu

Principe

Optimisation : trouver la meilleure des solutions possibles.
Programmation dynamique : stratégie d'optimisation.

Le principe est proche de diviser pour régner, mais les sous-problèmes ne sont plus nécessairement indépendants.

Rendu de monnaie

Problème : rendre $n = 45$ euros avec des pièces de 1, 7 et 10.
Optimisation : minimiser le nombre de pièces utilisées.

Rendu de monnaie

Problème : rendre $n = 45$ euros avec des pièces de 1, 7 et 10.
Optimisation : minimiser le nombre de pièces utilisées.

Si $r(n)$ est le nombre minimal de pièces à rendre, alors

$$r(n) = \min_{x \in \{1,7,10\}} r(n-x) + 1.$$

Rendu de monnaie

Problème : rendre $n = 45$ euros avec des pièces de 1, 7 et 10.
Optimisation : minimiser le nombre de pièces utilisées.

Si $r(n)$ est le nombre minimal de pièces à rendre, alors

$$r(n) = \min_{x \in \{1,7,10\}} r(n-x) + 1.$$

On est tenté de diviser pour régner :

- Diviser : 3 sous-problèmes $n-1$, $n-7$ et $n-10$;
- Régner : 3 appels récursifs $\rightsquigarrow$ 3 sous-solutions ;
- Combiner : minimum des sous-solutions.

Complexité

3 sous-problèmes guère plus petits $\rightsquigarrow$ complexité exponentielle ?

$$C(n) \geq C(n-1) + C(n-7) + C(n-10) \Rightarrow \exists \mu > 1,\ C(n) \geq c\mu^n.$$

Complexité

3 sous-problèmes guère plus petits $\rightsquigarrow$ complexité exponentielle ?

$$C(n) \geq C(n-1) + C(n-7) + C(n-10) \Rightarrow \exists \mu > 1,\ C(n) \geq c\mu^n.$$

Mais ces sous-problèmes se recouvrent largement : $\{r(k)\}_{k=1,\ldots,n}$.
Stocker chaque $r(k)$ une fois calculé $\rightsquigarrow$ complexité linéaire en n.

Complexité

3 sous-problèmes guère plus petits $\rightsquigarrow$ complexité exponentielle ?

$$C(n) \geq C(n-1) + C(n-7) + C(n-10) \Rightarrow \exists \mu > 1,\ C(n) \geq c\mu^n.$$

Mais ces sous-problèmes se recouvrent largement : $\{r(k)\}_{k=1,\ldots,n}$.
Stocker chaque $r(k)$ une fois calculé $\rightsquigarrow$ complexité linéaire en n.

Application : rendre 45 euros avec des pièces de 1, 7 et 10 euros.

En C

```
int rendu(int* tab, int n)
{
 if (n==1||n==7||n==10) tab[n]=1;
 if (tab[n]>0) return tab[n];
 int c=rendu(tab,n-1);
 if (n>7) c=min(c,rendu(tab,n-7));
 if (n>10) c=min(c,rendu(tab,n-10));
 tab[n]=c+1;
 return c+1;
}
```

Où `tab` est alloué dynamiquement en étant initialisé à 0 :

```
int* tab=calloc(n+1,sizeof(int));
```

Limites

Deux propriétés cruciales pour la programmation dynamique :

- Sous-optimalité : la solution se déduit des sous-solutions
  $\rightsquigarrow$ facilite la phase “combiner”.
- Recouvrement : un même sous-problème réapparaît souvent
  $\rightsquigarrow$ limite les appels récursifs via stockage des sous-solutions.

Limites

Deux propriétés cruciales pour la programmation dynamique :

- Sous-optimalité : la solution se déduit des sous-solutions
  $\rightsquigarrow$ facilite la phase "combiner".
- Recouvrement : un même sous-problème réapparaît souvent
  $\rightsquigarrow$ limite les appels récursifs via stockage des sous-solutions.

Ces propriétés ne sont pas toujours vérifiées :

- problème du voyageur de commerce
- problème du sac-à-dos
- ...

Ayez le polycopié de cours en TD/TP !

Il est disponible sur la page de Thomas Duyckaerts :
`www.math.univ-paris13.fr/~duyckaer/enseignement.html`

Et ces slides sur la page de Thomas Fernique :
`lipn.univ-paris13.fr/~fernique/teaching.html`