Informatique tronc commun
Tableaux et éléments

Corrigé

26 octobre 2007

1 Recherche du $k^{\text{ème}}$ élément d'un ensemble

Implémentation

```
tri_sel:=proc(T::array)
    local n,i,j,a;
    n:=rhs(op(2,eval(T)));
    for i from 1 to n do
        for j from i+1 to n do
            if T[j]<T[i] then
                a:=T[i];T[i]:=T[j];T[j]:=a
            fi
        od
    od
end;


median_paquets:=proc(T::array)
    local n,i,U,m,V,j;
    n:=rhs(op(2,eval(T)));
    m:=floor(n/5);
    U:=array(1..m);
    V:=array(1..5);
    for i from 1 to m do
        for j from 1 to 5 do
            V[j]:=T[5*(i-1)+j];
        od;
        tri_sel(eval(V));
        U[i]:=V[3];
    od;
    RETURN(eval(U));
end;

pivotage:=proc(T::array,pivot::integer)
    local n,i,j,a;
    n:=rhs(op(2,eval(T)));
    j:=1;
    for i from 1 to n do
        if T[i]<pivot then
            a:=T[j];
            T[j]:=T[i];
            T[i]:=a;
            j:=j+1
        fi
    od;
    RETURN(j)
end;

sous_tableau:=proc(T::array,a::integer,b::integer)
    local U,i;
    U:=array(1..1+b-a);
    for i from 1 to 1+b-a do
        U[i]:=T[a+i-1]
    od;
    RETURN(eval(U));
end;

keme:=proc(T::array,k::integer)
    local n,U,V,pivot,i;
    n:=rhs(op(eval(T)));
    print(n,k);
    if n<5
    then
        tri_sel(eval(T));
        RETURN(T[k]);
    else
        U:=median_paquets(eval(T));
        pivot:=keme(eval(U),ceil(n/10));
        i:=pivotage(eval(T),pivot);
        if i<=k then
            RETURN(keme(sous_tableau(T,i,n),k-i+1))
        else RETURN(keme(sous_tableau(T,1,i-1),k))
        fi
    fi
end;
```

Complexité Considérons un appel de `keme` sur un tableau T de taille n. On obtient le pivot x, et les tas $T_<$ (de taille $n_<$) et $T_>$ (de taille $n_>$). $n_<$ comme $n_>$ sont supérieurs à $3(n-4)/10$ (en tenant compte du fait que il y a un peut-être un paquet de moins de 5 éléments au bout que l'on a ignoré), donc inférieurs à $7(1+\epsilon)n/10$ (dès que n est assez grand). Le premier appel récursif (pour calculer x) a été fait sur une liste de taille $n/5$. On note $T(n)$ le temps maximal mis sur une liste de taille inférieure à n. On a $T(n) \leq T(7(1+\epsilon)n/10) + T(n/5) + Kn$ où K est une certaine constante. On montre alors par récurrence qu'il existe une constante K' telle que $T(n) \leq K'n$. Tout repose sur le fait que $7(1+\epsilon)/10 + 1/5 < 10$.

Optimalité On va montrer qu'un algorithme effectuant au plus $n-2$ comparaisons ne peut pas résoudre le problème. Supposons le contraire. On se donne un tableau T, sur lequel on fait tourner

l'algorithme. Il renvoit un résultat s. On considère le graphe G sur $\{1,\dots,n\}$ où l'on met une arête entre i et j si et seulement si l'algorithme compare T_i et T_j lorsqu'on le fait tourner sur T.

On montre par récurence qu'un graphe connexe (i.e. où il y a toujours un chemin joignant deux nœuds quelconques) sur n sommets doit avoir au moins $n-1$ arêtes. C'est vrai pour $n=2$. Supposons que cela soit vrai pour $n-1$ ($n \geq 3$), et prenons un graphe connexe à n sommets et m arêtes. Si tout sommet est adjacent à au moins deux arêtes, alors, en comptant de deux manières le nombre de paires (a,x) où x est un sommet, a une arête, et x est l'un des deux sommets de a, on obtient que $2a \geq 2n$. Donc, si $a < n$, il existe un sommet x adjacent à au plus une arête. G étant connexe avec plus de deux sommets x doit être adjacent à exactement une arête. Si l'on supprime x et son arête incidente, on optient un graphe toujours connexe avec $n-1$ sommets et $a-1$ arêtes, et l'hypothèse de récurence nous donne que $a \geq n-1$.

Donc le graphe G obtenu à partir de l'exécution de l'algorithme sur T n'est pas connexe. C'est à dire que l'on peut partitionner $\{1,\dots,n\}$ en deux parties non-vides A et B telles que pour tous $i \in A$ et $j \in B$ l'algorithme ne compare jamais T_i et T_j. Si il existe $i \in B$ tel que $T_i < s$ (et sinon, on procède similairement), on considère un tableau T' dans lequel $\forall (i,j) \in A^2 \cup B^2 T'_i < T'_j \iff T_i < T_j$, et $\forall (i,j) \in A \times B, T'_i < T'_j$. Si l'on fait tourner l'algorithme sur T' toutes les comparaisons effectuées donneront le même résultat, donc au final on obtiendra le même résultat s. Or si s était le $k^{\text{ème}}$ élément de T, il est maintenant au plus le $(k-1)^{\text{ème}}$ de T', d'où la contradiction.

2 Recherche du $k^{\text{ème}}$ élément de la réunion de deux tableaux triés

2.1 L'algorithme

Notons $s = |S|$ et $t = |T|$.
- Si $S_{\lceil S/2 \rceil} \leq T_{\lceil T/2 \rceil}$ et $\lceil S/2 \rceil + \lceil S/2 \rceil \leq k$, alors $S_{\lceil S/2 \rceil}$ est inférieur ou égal au $k^{\text{ème}}$ élément que l'on cherche.
- Si $S_{\lceil S/2 \rceil} \leq T_{\lceil T/2 \rceil}$ et $\lceil S/2 \rceil + \lceil S/2 \rceil > k$, alors $T_{\lceil S/2 \rceil}$ est strictement supérieur au $k^{\text{ème}}$ élément que l'on cherche.
- Les autres cas se traitent symétriquement.

Donc on peut éliminer la moitié de T ou de S. On recomence sur le reste, en réduisant k si l'on a éliminé une moitié inférieure de tableau. Quand l'un des deux tableaux n'a plus qu'un élément, on choisit entre cet élément et les $k^{\text{ème}}$ et $k-1^{\text{ème}}$ éléments de l'autre tableau.

À chaque étape, on coupe un des tableaux en deux, donc on fait $\Theta(\log|T| + \log|S|)$ étapes.

2.2 Implémentation

```
aux:=proc(S,s1,s2,T,t1,t2,k) local a,b;
    print(s1,s2,t1,t2,k);
    if s2=s1 then
        if t1−1+k>t2 then
            RETURN(max(T[t1−2+k],S[s1]))
        else
            if T[t1−1+k]>S[s1] then
                if k>1 then RETURN(max(T[t1−1+k−1],S[s1]))
                else RETURN(S[s1]) fi
            else RETURN(T[t1−1+k]) fi
        fi
    else
        if t2=t1 then
            if s1−2+k>s2 then
                RETURN(max(S[s1−2+k],T[t1]))
            else
                if S[s1−1+k]>T[t1] then
                    if k>1 then RETURN(max(S[s1−1+k−1],T[t1]))
                    else RETURN(T[t1]) fi
                else RETURN(S[s1−1+k]) fi
            fi
        else
            a:=ceil((s2−s1+1)/2);
            b:=ceil((t2−t1+1)/2);
            if (S[s1−1+a]<T[t1−1+b]) then
                if a+b < k then
                  RETURN(aux(eval(S),s1+a,s2,eval(T),t1,t2,k−a))
                else
                  RETURN(aux(eval(S),s1,s2,eval(T),t1,t1−1+b,k))
                fi
            else
                if a+b < k then
                  RETURN(aux(eval(S),s1,s2,eval(T),t1+b,t2,k−b))
                else RETURN(aux(eval(S),s1,s1−1+a,eval(T),t1,t2,k))
                fi
            fi
        fi
    fi
end;
```

```
keme:=proc(S,T,k) local s,t;
    s:=rhs(op(2,eval(S)));
    t:=rhs(op(2,eval(T)));
    RETURN(aux(eval(S),1,s,eval(T),1,t,k));
end;
```

3 Sous-suite commune maximale

3.1 Relation de récurrence

Les cas de base sont ceux où l'une des deux suites est vide, et le résultat est alors la suite vide.

Autrement, posons $_1 = a, {}'_1$ et $_2 = b, {}'_2$.

Si $a = b$, alors $_1 \cap {}_2 = a, {}'_1 \cap {}'_2$.

Sinon, $_1 \cap {}_2$ est la plus longue des deux suites $_1 \cap {}'_2$ et ${}'_1 \cap {}_2$.

3.2 Complexité de l'implémentation naïve

Si l'on implémente directement la récurrence ci-dessus, on a, (si les deux listes sont d'intersection vide, et en notant n la somme de leur taille) $T(n) = 2T(n-1)+O(1)$, d'où $T(n) = \Theta(2^n)$. Donc ça ne va pas.

4 Obtention d'une complexité temporelle quadratique

Si cet algorithme est exponentiel, c'est qu'il refait plein de fois la même chose. Pour l'améliorer, on va faire en sorte de partager des calculs entre les différentes branches. Si l'on y regarde de plus près, on voit que la procédure n'est jamais appelée que sur des suffixes de $_1$ et L_2. Donc il n'y a que $\mathcal{O}(n^2)$ instances du problème à traiter. Alors que l'algorithme naïf s'appelle $\Theta(2^n)$ fois, il retraite donc plein de fois le même problème. Pour l'améliorer, il suffit donc de mettre en mémoire les résultats que l'on calcule.

On note $m = |_1|$ et $n = |_2|$. Idéalement, à chaque fois que l'on a calculé un $'_1 \cap '_2$, on stocke le résultat dans la case $(m-|_1|, n-|_2|)$ d'une matrice. Lorsque l'on veut calculer $'_1 \cap '_2$, on commence par regarder dans la case correspondante de la matrice si le résultat n'a pas déjà été calculé. L'ennui, c'est que quand $_1$ et $_2$ commencent par le même nombre, on doit le mettre en tête de $_1 \cap _2$. Si l'on représente les sous-suites communes par des tableaux, on obtient alors un cout cubique en temps comme en mémoire. Dans un premier temps, on va ne calculer que la longueur des sous-suites communes.

Ensuite, une que tout est calculé, on peut construire la sous-suite croissante maximale simplement en refaisant le même calcul, sauf que l'on sait si l'on doit descendre en $(i+1, j)$ ou $(i, j+1)$ et n'a donc pas besoin d'essayer les deux.
Cela s'implémente ainsi:

```
aux:=proc(A,B,i,j,M)
    if M[i,j]>=0 then RETURN()
    else
        if A[i]=B[j] then
            aux(A,B,i+1,j+1,M);
            M[i,j]:=1+M[i+1,j+1]
        else
            aux(A,B,i+1,j,M);
            aux(A,B,i,j+1,M);
            M[i,j]:=max(M[i+1,j],M[i,j+1])
        fi;
        RETURN()
    fi
end;
```

```
lcs:=proc(A,B)
    local n,m,i,j,k,M,R;
    n:=rhs(op(2,eval(A)));
    m:=rhs(op(2,eval(B)));
    M:=array(1..n+1,1..m+1);
    for i from 1 to n+1 do
        for j from 1 to m+1 do
          if i=n+1 or j=m+1 then
            M[i,j]:=0
          else
            M[i,j]:= -1
          fi
        od;
    od;
    aux(A,B,1,1,M);
    R:=array(1..M[1,1]);
    i:=1;j:=1;k:=1;
    while k<=M[1,1] do
        if A[i]=B[j] then
            R[k]:=A[i];
            i:=i+1;
            j:=j+1;
            k:=k+1
        else
            if M[i+1,j] > M[i,j+1] then
                i:=i+1
            else
                j:=j+1
            fi;
        fi
    od;
    RETURN(eval(R));
end;
```

On notera (n_1, n_2) le problème dont l'on note la solution dans la case (n_1, n_2).

Ainsi, on ne calcule chaque (n_1, n_2) qu'une fois. De plus, on ne relit sa valeur qu'au plus deux fois. En effet, il n'y a qu'au plus trois manières d'atterrir dessus : depuis (n_1-1, n_2), (n_1, n_2-1) ou (n_1-1, n_2-1). Donc si l'on avait besoin de relire plus de deux fois (n_1, n_2), cela voudrait dire que l'on descendrait plusieurs fois à travers l'un de ses pères. Ce qui ne se peut pas.

Donc, on ne fait que $\mathcal{O}(1)$ opérations par sous-problème, d'où au final $\mathcal{O}(nm)$. L'espace mémoire est donc en $\mathcal{O}(nm)$, et il est même en $\Theta(nm)$ vu que l'on alloue un tableau de taille nm.