Algorithmique et programmation en Maple
Rapple sur les algorithmes de tris

1 Tri par sélection

C'est celui qui vient naturellement à l'esprit quand on réfléchit un peu : on sélectionne le plus petit élément et on le met de coté, puis on sélectionne le plus petit de ceux qui restent et ainsi de suite ... En voici une implémentation :

```
insertion_sort:=proc(t) local i,j,n,tmp;
    n:=rhs(op(2,eval(t)));
    for i from 1 to n do
        for j from i to n do
            if t[i]>t[j] then
                tmp:=t[i];t[i]:=t[j];t[j]:=tmp;
            fi;
        od;
    od;
end;
```

À la i$^{\text{ème}}$ itération extérieure, on parcourt les $n-i$ cases restantes du tableau, donc la complexité est $\Theta(\sum_{i=1}^{n}(n-i))$, ie $\Theta(n^2)$.

2 Tri par insertion

Celui-ci est un peu plus subtil : on itère sur une variable i qui varie de 1 à n. Au début de la $i^{\text{ème}}$ itération, les $i-1$ premières cases sont triées, mais ne forment pas forcément les $i-1$ premières cases du tableau final (dans les $n-i$ cases restantes, il peut rester des nombres à intercaler). Le corps de la $i^{\text{ème}}$ itération consiste à placer le contenu de la $i^{\text{ème}}$ case de sorte que les i premières cases soient triées. Pour ce faire, il suffit de l'échanger avec son prédécesseur tant que ce dernier est plus grand. Implémentation :

```
selection_sort:=proc(t) local i,j,n,tmp;
    n:=rhs(op(2,eval(t)));
    for i from 2 to n do
        tmp:=t[i];
        for j from i-1 by -1 to 1 while t[j]>tmp do
            t[j+1]:=t[j]
        od;
        t[j+1]:=tmp
    od;
end;
```

La complexité est plus dure à déterminer, elle dépend fortement de la distribution des éléments dans le tableau. Si le tableau est déjà trié, la boucle interne ne fait qu'un tour à chaque fois et l'on fait juste $\mathcal{O}(n)$ opérations. En revanche, si le tableau est trié à l'envers, la boucle interne recule à chaque fois jusqu'au début du tableau, et cela donne donc du $\Theta(n^2)$. La seule chose simple que l'on puisse dire c'est que la complexité au pire est $\Theta(n^2)$. De plus, on peut montrer que la moyenne *sur toutes les permutations possibles* du nombre d'opérations est en $\Theta(n^2)$.

2.0.1 Tri fusion

On coupe le tableau en deux morceaux de même taille (à 1 près), on trie les deux morceaux en appliquant la même méthode, puis l'on combine les résultats via la procédure `fus` de la section précédente. Cela donne un tri en $\Theta(n \ln n)$.

Première étape : fusion de deux tableaux triés en un tableau trié. À chaque étape on se ramène à fusionner deux tableaux dont l'un est de la même taille qu'avant et l'autre a vu sa taille diminuée de 1. C'est linéaire.

```
fus:=proc(t1,t2) local t,n,m,i;
    m:=rhs(op(2,eval(t1)));
    n:=rhs(op(2,eval(t2)));
    t:=array(1..n+m);
    while m>0 and n>0 do
        if t1[m]<t2[n] then t[n+m]:=t2[n];n:=n-1
        else t[n+m]:=t1[m];m:=m-1 fi
    od;
    if m>0 then for i from m to 1 by -1 do t[i]:=t1[i] od
    else for i from n to 1 by -1 do t[i]:=t2[i] od fi;
    RETURN(eval(t));
end;
```

Seconde étape : on effectue deux appels récursifs d'une préocédure linéaire à chaque appel de la procédure de fusion : l'algorithme est en $\Theta(n \ln n)$.

```
fsort := proc(t)
local n,m,res,t1,t2;
    n := rhs(op(2,eval(t)));
    if (n=0) then res:=[];
    elif (n=1) then res:=t;
    else
      m := floor(n/2);
      t1:=fsort(vector([seq(t[i],i=1..m)]));
      t2:=fsort(vector([seq(t[i],i=(m+1) ..n)]));
      res:= fus(eval(t1),eval(t2));
    fi; eval(res);
end;;
```

2.0.2 Tri rapide

C'est encore un diviser pour régner. Mais au lieu de diviser aveuglément comme dans le tri fusion, on choisit un pivot (par exemple le premier élément) parmi les éléments du tableau, puis l'on sépare les éléments du tableau plus petits que le pivot de ceux qui sont plus grand, et l'on trie les deux tas récursivement. La fusion est quant à elle triviale. Implémentation :

```
split:=proc(t,i,j) local k, mid, tmid;
    mid:=i;
    tmid:=t[i];
    for k from i+1 to j do
        if t[k]<=tmid then
            t[mid]:=t[k];t[k]:=t[mid+1];
            mid:=mid+1 fi
    od;
    t[mid]:=tmid;
    RETURN(mid);
end;

qsort:=proc(t,i,j,leq) local mid;
    if j<=i
    then RETURN();
    else mid:=split(t,i,j,leq);
        qsort(t,i,mid-1);
        qsort(t,mid+1,j) fi
end;
```

Là encore, la complexité dépend beaucoup de la permutation initiale : si le tableau est initialement trié, on obtient du $\Theta(n^2)$.
Pour formaliser cette intuition, il faut faire un calcul en moyenne sur toutes les permutations possibles, et on admettra que l'on obtient bien du $\Theta(n \ln n)$. Cela veut dire que les cas où l'algorithme dégénère en $\Theta(n^2)$ sont assez rares, et que l'on peut donc à priori l'utiliser sans problème. C'est effectivement l'algorithme le plus employé car il se révèle être plus rapide en pratique que les tris « stables » comme le tri fusion (l'une des raisons pour cela est qu'il n'a besoin que de peu de mémoire : juste de quoi se souvenir des appels récursifs en attente, soit $\mathcal{O}(\ln n)$ en moyenne, alors que le tri fusion nécessite un tableau supplémentaire de taille n).