Partiel d'Algorithmique n° 1

Corrigé succinct du partiel du 27 octobre 2000

Exercice 1

1. **Exécution du programme** Le programme affiche **exactement** la séquence suivante: 1 2 3 5 10 25

   D'une manière générale, on peut dire que le programme sert à trier un ensemble d'entiers par ordre croissant.

2. **Recherche du plus petit élément dans une liste** Si on appelle n la longeur de la liste, le programme va parcourir **exactement** toute la liste, et effectuer des tâches de temps constant $O(1)$ pour chaque élement de la liste (comparaison plus avancée dans la liste, et éventuellement une allocation). La complexité de cette fonction va donc être:$\sum_{i=1}^{n} O(1) = O(n)$.

3. **Tri d'une liste** L'idée générale est qu'au début, on va extraire le plus petit élément d'une liste de longeur n, puis d'une liste de longeur $n-1$...etc... Ainsi la complexité finale va être le résultat du calcul: $C = \sum_{i=1}^{n} O(i) = O(n^2)$ On pourra remarquer que l'insertion prend un temps $O(n)$ qui n'est pas le temps critique dans l'algorithme.

4. **Tri d'une liste optimisée** Ici l'insertion **et** la suppression prennent un temps $O(\log n)$ où n représente la longeur de la liste. Ainsi la complexité sera:

   $$C = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n)).$$

   Cette complexité est la meilleure qu'il puisse être théoriquement possible d'obtenir pour un tri.

Exercice 2

```
typedef int repbinaire;
typedef int bit;

repbinaire reste(repbinaire n){
  return(n/10);
}

bit premier(repbinaire n){
  return(n-(reste(n)*10));
}

int evaluer(repbinaire n){
  if (reste(n)==0)
     {return(premier(n));}
  else {return(premier(n)+2*evaluer(reste(n)));}
}
```

```
repbinaire somme(repbinaire n1,repbinaire n2){
  return(somme_avec_retenue(n1,n2,0));
}

repbinaire somme_avec_retenue(repbinaire n1,repbinaire n2,bit retenue)
{bit p1,p2;
 bit res_p, ret_p;

 p1=premier(n1);
 p2=premier(n2);
 res_p=bitResultat(p1,p2,retenue);
 ret_p=propagationRetenue(p1,p2,retenue);
 if ((reste(n1)==0) && (reste(n2)==0))
     {return(res_p+10*ret_p);}
     else return(res_p+10*somme_avec_retenue(reste(n1),reste(n2),ret_p));
}
```

Exercice 3

1. Insertion en queue de liste :

```
void insere (int i, LISTE *lptr)
{ LISTE tmp;
  tmp = (LISTE) malloc (sizeof(ELEMENT));
  tmp->valeur = i;
  if (*lptr == NULL) {
    tmp->suivant = tmp;
    *lptr = tmp;
  }
  else {
    tmp->suivant = (*lptr)->suivant;
    (*lptr)->suivant = tmp;
    *lptr = tmp;
  }
}
```

   Le nombre d'instructions exécutées est indépendant de la longueur de la liste. La complexité de `insere` et donc $\theta(1)$.

2. Suppression en tête : attention au cas particulier de la longueur 1 !

```
bool supprime_tete (LISTE *lptr, int *x)
{
  LISTE tmp;
  if (*lptr == NULL) return false;
  else {
    if ((*lptr)->suivant == *lptr) { /* la longueur 1 */
      *x = (*lptr)->valeur;
      free (*lptr); *lptr = NULL;
    }
    else {
      *x = (*lptr)->suivant->valeur;
      tmp = (*lptr)->suivant->suivant;
      free ((*lptr)->suivant);
      (*lptr)->suivant = tmp;
```

```
    }
    return true;
  }
}
```

3. ```
   int gagnant (LISTE *joueurs)
   /* on suppose que la liste est non vide */
   {
     int j1,j2;
     while (*joueurs != (*joueurs)->suivant) {
       supprime_tete (joueurs, &j1);
       supprime_tete (joueurs, &j2);
       if (j1>j2) { insere (j1, joueurs); }
       else { insere (j2, joueurs); }
     }
     return ((*joueurs)->valeur);
   }
   ```

   Les deux fonctions `insere` et `supprime_tete` sont de complexité $\theta(1)$. Le corps du `while` est exécuté $n-1$ fois pour un liste de n éléments (à chaque exécution un élément est supprimé). La complexité de gagnant est donc $\theta(n)$.

4. Si on positionne le pointeur sur la tête de la liste alors les deux fonctions `insere` et `supprime_tete` deviennent linéaire. Pour `insere`, il faut chercher le dernier de la liste pour faire l'insertion au bon endroit ; pour `supprime_tete`, il faut également chercher le dernier élément pour mettre à jour son champ `suivant`. On remarque que la complexité de `gagnant` devient alors quadratique.

Exercice 4

1. Structure: une liste chaînée. Insertions: on les fait en tête de la liste, en $\Theta(1)$. Supressions: on décide (arbitrairement) de supprimer l'élément en tête de liste, ce qui se fait également en $\Theta(1)$ d'après le cours.

2. Avec une liste chaînée, pour les suppressions rien de changé, en revanche pour insérer un élément il faut commencer par le rechercher pour savoir s'il est déjà présent dans la liste. La recherche prenant $\Theta(n)$ pour une liste de longueur n, la complexité de cette implémentation est maintenant $\Theta(n)$ pour les insertions et $\Theta(1)$ pour les suppressions.

3. On pourrait envisager d'utiliser un tableau t de taille $N+1$ avec $t[i]$ égal à vrai ou faux selon que i est déjà présent ou non dans le sac. Ainsi on pourrait en temps constant décider si i est déjà présent, et l'insére reviendrait à mettre $t[i]$ à vrai, ce qui prendrait temps constant. Problème: comment faire une supression avec cette implémentation ? Par exemple, s'il n'y a que 10 éléments dans le sac, il n'y a donc que 10 entrées du tableau qui sont à “Vrai”. Comment trouver l'une d'entre elles ? A priori cela exige de parcourir le tableau t, en $\Theta(N)$. Cette solution ne convient donc pas.

   La bonne solution consiste à utiliser à la fois un tableau de booléens et une liste chaînée. Insertion de i: on vérifie que $t[i] = faux$, on le met à vrai et on insère également i dans la liste chaînée. Suppression d'un élément arbitraire: soit i l'élément en tête de liste. On le supprime de la liste et on met $t[i]$ à faux. Clairement ces opérations prennent temps constant.
   ```