IN 101 - TD 04
30 septembre 2011

Corrigé

ENSTA ParisTech

Solution 1 : Algorithme glouton pour le rendu de monnaie

```
1  int main(int argc, char** argv) {
2    int s[9] = {500, 200, 100, 50, 20, 10, 5, 2, 1};
3    int total,i;
4    int first = 1;
5    if (argc > 1) {
6      total = atoi(argv[1]);
7    } else {
8      printf("Entrez la somme à atteindre en argument.\n");
9      return 1;
10   }
11   i=0;
12   while ((total != 0) && (i < 9)) {
13     if (total < s[i]) {
14       i++;
15     } else {
16       if (first) {
17         printf("%d = %d", total, s[i]);
18         first = 0;
19       } else {
20         printf(" + %d",s[i]);
21       }
22       total -= s[i];
23     }
24   }
25   if (total != 0) {
26     printf(" + ...\nEchec du rendu avec l'algorithme glouton.\n");
27   } else {
28     printf("\nRendu réussi !\n");
29   }
30   return 0;
31 }
```

Solution 2 : Calcul de coefficients binomiaux

2.a] Code des fonctions `factoriel` et `binom1`

```
1  int factoriel(int n) {
2    int i, res;
3    res = 1;
4    for (i=2; i<n+1; i++) {
5      res *= i;
6    }
7    return res;
```

```
}
int binom1(int n, int p) {
  return factoriel(n)/factoriel(p)/factoriel(n-p);
}
```

2.b] $100! > 2^{32}$, donc 100! ne tient pas sur un mot machine et la fonction `factoriel` retourne donc les 32 bits de poids faible de l'écriture binaire de 100!. Or 100! contient beaucoup de fois le facteur 2, suffisamment pour être multiple de 2^{32}. Du coup `factoriel(100)` retourne 0, et `factoriel(97)` aussi. On a donc une division par 0...

2.c] Code de la fonction `binom2`

```
int binom2(int n, int p) {
  if ((p==0) || (n==p)) {
    return 1;
  } else {
    return binom2(n-1,p-1) + binom2(n-1,p);
  }
}
```

2.d] La fonction `binom2` a une complexité égale à la valeur qu'elle retourne : le résultat est calculé en faisant une somme de 1. Les grands coefficients binomiaux sont donc longs à calculer car certains calculs sont refaits beaucoup de fois.

2.e] Code de la fonction `binom3` et le `main` qui va avec

```
int tab[101][101];
int binom3(int n, int p) {
  if (tab[n][p] == 0) {
    if ((p==0) || (n==p)) {
      tab[n][p] = 1;
    } else {
      tab[n][p] = binom3(n-1,p-1) + binom3(n-1,p);
    }
  }
  return tab[n][p];
}
int main() {
  int i,j;
  for (i=0; i<101; i++) {
    for (j=0; j<101; j++) {
      tab[i][j] = 0;
    }
  }
  printf("%d\n",binom3(100,6));
  return 0;
}
```

Cette fonction va calculer une et une seule fois tous les coefficients binomiaux $\binom{i}{j}$ pour $j \in [0; p]$ et $i \in [j; n-p+j]$. On calcule donc une sorte de parallélogramme dont chaque coefficient se calcule en $O(1)$. La complexité totale est donc $O(np)$.

2.f] Code des fonctions `gcd` (calcul du pgcd de deux entiers) et `binom4`

```
1  int gcd(int a, int b) {
2    if (b == 0) {
3      return a;
4    } else {
5      return gcd(b, a%b);
6    }
7  }
8
9  int binom4(int n, int p) {
10   int res, i,a;
11   if ((n-p) < p) {
12     return binom4(n,n-p);
13   }
14   res = 1;
15   for (i=0; i<p; i++) {
16     a = gcd(n-i,i+1);
17     res /= (i+1)/a;
18     res *= (n-i)/a;
19   }
20   return res;
21 }
```

Solution 3 : Exponentiation binaire

3.a] L'algorithme naïf permettant de calculer a^b effectue $(b-1)$ multiplications successives par a. Sa complexité est donc $\Theta(b)$ multiplications, si on considère une multiplication entre deux entiers comme une opération élémentaire.

Complexité en Opérations Binaires

On peut aussi s'intéresse à la complexité en termes d'opérations « bit à bit » (aussi appelées opérations binaires). On voit que la complexité de la multiplication de x par y est en $\Theta(\log(x) \times \log(y))$ (par exemple en utilisant l'algorithme que l'on utilise quand on pose une multiplication à la main). La complexité de l'algorithme naïf en termes d'opérations binaires est donc :

$$\begin{aligned}
&\Theta(\log(a) \times \log(a) + \log(a^2) \times \log(a) + \ldots + \log(a^{b-1}) \times \log(a)) \\
&\quad = \Theta\left(log(a)^2 \times \sum_{i=1}^{b-1} i\right) \\
&\quad = \Theta\left(\log(a)^2 \times \frac{b(b-1)}{2}\right) \\
&\quad = \Theta\left(\log(a)^2 \times b^2\right).
\end{aligned}$$

3.b] Afin de décomposer un entier b en base 2, on peut commencer par calculer le bit de poids faible b_0. Si $b_0 = 0$, alors b est pair et si $b_0 = 1$, b est impair. Par conséquent, b_0 est le reste de la division euclidienne de b par 2. Le même raisonnement appliqué à $\lfloor \frac{b}{2} \rfloor$ va permettre de calculer b_1 et ainsi de suite. On obtient l'algorithme suivant :

```c
void base2(int b) {
  while (b > 0) {
    /* la parité de b est son "et" bit à bit avec 1 */
    printf("%d\n", b&1);
    /* la division par 2 est un décalage à droite de 1 bit */
    b = b >> 1;
  }
}
```

En C, les opérations binaires sont très simples et rapides (car les machines travaillent en binaire). Si on voulait écrire une décomposition dans une autre base il faudrait utiliser les opérations *modulo* et *diviser* à la place du *et* et du décalage.

Le calcul de la complexité de cet algorithme est très simple : il suffit de calculer le nombre de fois que la boucle `while` est répétée. À chaque étape on fait un décalage à droite (une division par deux), donc la complexité correspond exactement au nombre de bits nécessaire pour écrire b en base 2, c'est-à-dire $\log_2(b) + 1$. La complexité de l'algorithme est donc $\Theta(\log(b))$.

3.c] On utilise l'équation $a^b = a^{\left(\sum_{i=0}^{k} b_i \times 2^i\right)} = \prod_{i=0}^{k} \left(a^{(2^i)}\right)^{b_i}$. On va donc mettre en table tous les $a^{(2^i)}$ et multiplier ceux pour lesquels b_i vaut 1. On obtient l'algorithme suivant (on suppose que k est connu et que $b < 2^k$) :

```c
int binary_exp(int a, int b) {
  int tab[k];
  int i,res;
  /* on initialise la table */
  tab[0] = a;
  for (i=1; i<k; i++) {
    tab[i] = tab[i-1]*tab[i-1];
  }
  /* on utilise la table pour le calcul */
  res = 1;
  for (i=0; i<k; i++)
    if ((b & (1<<i)) != 0) {
      res = res*tab[i];
    }
  }
  return res;
}
```

Cet algorithme commence par calculer les `tab[i]` $= a^{(2^i)}$ avant de multiplier ceux pour lesquels le bit associé b_i vaut 1. La complexité en fonction de k est facile à calculer. En effet, la première boucle effectue k élévations au carré de nombres entiers, et la seconde au plus k multiplications entières ; k étant de l'ordre de $\log(b)$, on obtient donc $\Theta(\log(b))$ multiplications entières. De plus, la première étape ne dépendant que de la taille de l'exposant, cette complexité est donc valable que ce soit en moyenne ou dans le cas le pire. Concernant la complexité spatiale, il est nécessaire de stocker $\Theta(\log(b))$ entiers.

3.d] Il est facile d'éviter le stockage des `tab[i]` en utilisant les valeurs au fur et à mesure de leur calcul dans la fonction précédente. Il est également possible d'intégrer directement la

décomposition en base 2, un peu comme dans la version récursive du TD03, mais en itératif. Cela donne le code suivant :

```
1  int binary_exp2(int a, int b) {
2    int res = 1;
3    while (b > 0) {
4      if (b & 1) {
5        res = res*a;
6      }
7      a = a*a;
8      b = b >> 1;
9    }
10   return res;
11 }
```

Cet algorithme a toujours la même complexité de $\Theta(\log(b))$ multiplications, mais a une complexité spatiale de $\Theta(1)$ entiers.

Complexité en Opérations Binaires

Si on regarde le nombre d'opérations binaires tout n'est pourtant pas si rose... En effet, le coût de la dernière élévation au carré est le coût d'une multiplication de deux entier de $\frac{b}{2}\log_2(a)$ *bits, soit* $\Theta(b^2\log(a)^2)$*. Au final, même si on ne fait plus que* $\log(b)$ *multiplications au lieu de b, la complexité binaire n'est pas meilleure que pour l'algorithme naïf. L'exponentiation binaire est donc surtout intéressante quand le coût d'une multiplication est constant tout au long de l'algorithme : c'est ce qu'il se passe en cryptographie...*

Solution 4 : Application à la cryptographie

4.a] L'algorithme précédent s'adapte très simplement en effectuant des réductions modulaires dès que possible (et surtout pas à la fin du calcul uniquement).

```
1  int modular_exp(int m, int e, int n) {
2    int c = 1;
3    while (e > 0) {
4      if (e & 1) {
5        c = c*m % n;
6      }
7      m = m*m % n;
8      e = e >> 1;
9    }
10   return c;
11 }
```

4.b] Avec des exposants de 1024 bits, on effectue 1024 élévations au carré et de l'ordre de 512 multiplications en moyenne (selon le nombre de bits valant 1 dans l'exposant). Il ne s'agit bien sûr plus ici d'opérations sur des `int` (limités à 32 ou 64 bits), mais d'opérations plus coûteuses sur des grands entiers. Un chiffrement nécessite donc de l'ordre de 1500 multiplications modulaires, ce qui est important, mais réalisable sur une carte à puce, par exemple. Notons que pour de telles tailles l'algorithme naïf nécessite de l'ordre de 2^{1024} multiplications...

4.c] On a $e = 65537 = 2^{16} + 1$. L'avantage de cet exposant est de ne nécessiter que deux multiplications et non 8 comme c'est le cas en moyenne. Il faut donc 16 élévations au carré plus deux multiplications soit 18 opérations modulaires, au lieu de 24 en moyenne. De plus 65537 est un nombre premier.