IFT2125-6001 TA: Maëlle Zimmermann

Démonstration 9

1

Question: Un circuit $n - tally$ est un circuit qui prend n bits en entrée et produit $1 + \lfloor \log n \rfloor$ bits en sortie. Il compte en binaire le nombre de bits égaux à 1 dans l'entrée. Par exemple, si $n = 9$ et l'entrée est 011001011, alors il y a 5 bits égaux à 1, et la sortie est 0101 (5 en binaire). Un $(i, j) - adder$ est un circuit qui prend un nombre m de i bits et un nombre n de j bits en entrée. Il calcule $m + n$ en binaire sur $1 + \max(i, j)$ bits de sortie. Par exemple, si l'entrée est $m = 101$ et $n = 10111$ ($i = 3$, $j = 5$), la sortie est la somme des deux nombres, soit 011100.

Il est toujours possible de construire un $(i, j) - adder$ à partir d'exactement $\max(i, j)$ $3 - tallies$. En effet, additionner $m + n$ revient à compter pour chaque position k le nombre de bits égaux à 1 parmi le $k$ème bit de m, le $k$ème bit de n, et l'éventuel bit de retenue. Comme le calcul doit être fait pour $\max(i, j)$ positions k nous avons besoin de $\max(i, j)$ $3 - tallies$.

1. Utiliser des $3 - tallies$ et des $(i, j) - adders$ afin de construire un $n - tally$ efficace.
2. Donner une récurrence (avec condition initiales) qui décrit le nombre de $3 - tallies$ nécessaires pour construire le $n - tally$, incluant les $3 - tallies$ qui font partie des $(i, j) - adders$.
3. Résoudre la récurrence exactement.

Solution:

1. Nous construisons un $n - tally$ de façon récursive. Lorsque $1 \leq n \leq 3$, il suffit d'utiliser un $3 - tally$. Lorsque $n > 3$ nous divisons l'entrée en deux en construisons en $\lceil n/2 \rceil - tally$ et un $\lfloor n/2 \rfloor - tally$, comptant le nombre de bits égaux à 1 dans chaque moitié de l'entrée. Le résultat de ces deux $tallies$ est sommé par un $(i, j) - adder$ où $i = 1 + \lfloor \log \lceil n/2 \rceil \rfloor$ et $j = 1 + \lfloor \log \lfloor n/2 \rfloor \rfloor$.

2. Soit $t(n)$ le nombre de $3-tallies$ utilisés afin de construire un $n-tally$ dans la construction donnée en (1). Lorsque $1 \leq n \leq 3$, un seul $3-tally$ est utilisé. Lorsque $n > 3$ le nombre de $3-tallies$ utilisés est $t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor)$, plus le nombre de $3-tallies$ utilisés afin de construire le $(i,j)-adder$, c'est-à-dire $\max(i,j)$. Comme $i = 1 + \lfloor \log \lceil n/2 \rceil \rfloor$ et $j = 1 + \lfloor \log \lfloor n/2 \rfloor \rfloor$, nous obtenons

$$t(n) = \begin{cases} 1 & \text{si } 1 \leq n \leq 3, \\ \underbrace{t(\lceil \frac{n}{2} \rceil)}_{\lceil \frac{n}{2} \rceil - tally} + \underbrace{t(\lfloor \frac{n}{2} \rfloor)}_{\lfloor \frac{n}{2} \rfloor - tally} + \underbrace{1 + \lfloor \log \lceil \frac{n}{2} \rceil \rfloor}_{(i,j)-adder} & \text{si } n > 3 \end{cases} \tag{1}$$

3. Posons $s(i) = t(2^i)$, alors nous avons

$$s(i) = \begin{cases} 1 & \text{si } 0 \leq i \leq 1, \\ 2s(i-1) + i & \text{si } i > 1 \end{cases}$$

Le polynôme charactéristique de la récurrence s est $p(x) = (x-2)(x-1)^2$ et ainsi $s(i) = c_1 2^i + c_2 + c_3 i$. En résolvant le système

$$\begin{array}{lclclcl} c_1 & + & c_2 & + & & = & 1 \\ 2c_1 & + & c_2 & + & c_3 & = & 1 \\ 4c_1 & + & c_2 & + & 2c_3 & = & 4 \end{array}$$

nous obtenons $c_1 = 3, c_2 = -2$ et $c_3 = -3$. Ainsi, $s(i) = 3 \cdot 2^i - 3i - 2$ et donc $t(n) = s(\log n) = 3n - 3\log n - 2$ lorsque n est une puissance de 2.

Nous avons donc $t(n) \in \Theta(n : n \text{ est une puissance de } 2)$. Puisque $t(n)$ est éventuellement non décroissante (on peut le démontrer), nous concluons par la règle de l'harmonie que $t(n) \in \Theta(n)$.

Alternativement, si on cherche simplement à obtenir l'ordre de t et non sa forme exacte, on peut utiliser le théorème vu en classe (premier cas). Nous avons $a = 2, b = 2$, et $f(n) = \log(n) \in O(n^{\log 2 - \epsilon})$ en prenant n'importe quel ϵ suffisamment petit (par exemple 0.1). On en conclut également que $t(n) \in \Theta(n : n \text{ est une puissance de } 2)$.

2

Question: Supposons que nous avons accès aux algorithmes suivants:

- `mult_k1`: multiplie un polynôme de degré k avec un polynôme de degré 1 en un temps $O(k)$,

- `mult_kk`: multiplie deux polynômes de degré k en un temps $O(k \log k)$.

Soient $z_1, ..., z_d \in \mathbb{Z}$. Donner un algorithme efficace qui calcule l'unique polynôme $p(n) = a_0 + a_1 n + ... + a_d n^d$ tel que $a_d = 1$ et $p(z_1) = ... = p(z_d) = 0$. Notez que nous représenterons un polynôme $a_0 + a_1 n + ... + a_d n^d$ par le tableau $[a_0, a_1, ..., a_d]$. Analyser l'efficacité de l'algorithme.

Solution: Il suffit de calculer le polynôme $p(n) = (n - z_1)(n - z_2)...(n - z_d)$ récursivement en découpant successivement la liste $z_1, ..., z_d$ en deux. Voici un tel algorithme:

```
def zeros(z):
  if len(z) == 0:
    return [1]
  elif len(z) == 1:
    return [-z[0], 1]
  else:
    m = len(z) // 2
    q = zeros(z[:m])
    r = zeros(z[m:2*m])
    # len(z) pair / even
    if len(z) % 2 == 0:
      return mult_kk(q, r)
    # len(z) impair / odd
    else:
      s = zeros(z[-1:])
      return mult_k1(mult_kk(q, r), s)
```

Le temps d'exécution de `zeros` est décrit par la récurrence suivante:

$$t(d) = \begin{cases} 1 & \text{si } d \leq 1, \\ 2t(\lfloor \frac{d}{2} \rfloor) + f(\lfloor \frac{d}{2} \rfloor) & \text{si } d > 1 \text{ et est pair,} \\ 2t(\lfloor \frac{d}{2} \rfloor) + t(1) + f(\lfloor \frac{d}{2} \rfloor) + g(d-1) & \text{si } d > 1 \text{ et est impair} \end{cases}$$

où $f(d) \in O(d \log d)$ et $g(d) \in O(d)$. Ainsi,

$$t(d) \in \begin{cases} 1 & \text{si } d \leq 1, \\ 2t(\lfloor \frac{d}{2} \rfloor) + O(d \log d) & \text{si } d > 1. \end{cases}$$

Appliquons le théorèmes sur les récurrences vu en classe. Nous avons $a = 2, b = 2$ et $f(d) = d \log d$. Posons $\epsilon = 1$. Puisque $f(d) \in O(d \log d) = O(d^{\log_b a} (\log d)^{\epsilon})$, nous concluons que $t(d) \in O(d^{\log_b a} (\log d)^{\epsilon + 1}) = O(d (\log d)^2)$.

3

Question: Soit la matrice $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Que se passe-t-il quand on élève A à la puissance 2? Et à la puissance n? Sur ce principe, construire un algorithme diviser-pour-régner pour calculer le $n^{\text{ème}}$ élément de la séquence de Fibonacci. Analyser l'efficacité de l'algorithme avec la notation O en supposant 1) que les opérations arithmétiques ont un coût constant, puis 2) que multiplier deux entiers de taille s et q prend un temps dans $\Theta(sq^{\alpha-1})$ si $s \geq q$. Ici α est une constante qui dépend de l'algorithme utilisé pour faire le produit. Par exemple, pour l'algorithme efficace vu en cours, on a $\alpha = \log_2 3$. On rappelle que la taille (en bits) du $n^{\text{ème}}$ nombre de Fibonacci est dans $\Theta(n)$.

Solution: On remarque d'abord que

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} F^{n-1} & F^n \\ F^n & F^{n+1} \end{pmatrix}.$$

Ainsi il suffit d'implémenter un algorithme diviser-pour-régner qui calcule la $n^{\text{ème}}$ puissance de la matrice A. La méthode est similaire à l'algorithme expoDC qui calcule la $n^{\text{ème}}$ puissance d'un nombre a dans BB section 7.7. En effet:

$$A^n = \begin{cases} A & \text{si } n = 1, \\ (A^{n/2})^2 & \text{si } n > 1 \text{ et est pair} \\ AA^{n-1} & \text{si } n > 1 \text{ et est impair} \end{cases}$$

Soit $T(n)$ la récurrence qui décrit le temps de l'algorithme, et soit $M(s,q)$ le temps pour multiplier deux entiers de taille s et q où $s \geq q$.

Si n est pair, la matrice à élever au carré est $\begin{pmatrix} F^{n/2-1} & F^{n/2} \\ F^{n/2} & F^{n/2+1} \end{pmatrix}$. Cela prend 8 multiplication de nombres de taille maximum $\frac{n}{2}+1$. Ainsi élever la matrice au carré prend un temps borné par $8M(\frac{n}{2}+1, \frac{n}{2}+1)$.

Si n est impair, il faut effectuer le produit entre $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ et $\begin{pmatrix} F^{n-2} & F^{n-1} \\ F^{n-1} & F^n \end{pmatrix}$. Cela prend 8 multiplications d'un nombre de taille 1 avec un nombre de taille maximum n. Donc effectuer le produit des deux matrices prend un temps borné par $8M(n,1)$. On obtient:

$$T(n) \leq \begin{cases} 0 & \text{si } n = 1, \\ T(\frac{n}{2}) + 8M(\frac{n}{2}+1, \frac{n}{2}+1) & \text{si } n > 1 \text{ et est pair,} \\ T(n-1) + 8M(n,1) & \text{si } n > 1 \text{ et est impair} \end{cases}$$

Il faut donc trouver une borne qui s'applique au cas pair et impair. Notons que si n est impair, nous avons:

$$\begin{aligned} T(n) &\leq T(n-1) + 8M(n,1) \\ &= T(\frac{n-1}{2}) + 8M(\frac{n-1}{2}+1, \frac{n-1}{2}+1) + 8M(n,1)) \end{aligned}$$

Cela implique que $\forall n > 1$,

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + 8M(\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 1) + 8M(n,1)).$$

Analysons l'efficacité de l'algorithme selon chaque supposition.

1) Si les opération arithmétiques et en particulier les multiplications ont un coût constant (notons que cette supposition est peu réaliste), on obtient

$$T(n) \in \begin{cases} 1 & \text{si } n = 1, \\ T(\lfloor \frac{n}{2} \rfloor) + O(1) & \text{si } n > 1. \end{cases}$$

Ainsi on obtient en appliquant le théorème sur les récurrences vu en classe (cas 3 avec $\epsilon = 0$) que $T(n) \in O(\log n)$.

2) En revanche si $M(s,q) \in \Theta(sq^{\alpha-1})$, alors $8M(\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 1) \in \Theta((\lfloor \frac{n}{2} \rfloor + 1)(\lfloor \frac{n}{2} \rfloor + 1)^{\alpha-1}) = \Theta((\lfloor \frac{n}{2} \rfloor + 1)^{\alpha}) = \Theta(n^{\alpha})$ et $8M(n,1) \in \Theta(n1^{\alpha-1}) = \Theta(n)$. Comme $\alpha > 1$, nous avons

$$T(n) \in \begin{cases} 1 & \text{si } n = 1, \\ T(\lfloor \frac{n}{2} \rfloor) + O(n^{\alpha}) & \text{si } n > 1. \end{cases}$$

Appliquons le théorème sur les récurrences vu en classe (cas 2). Nous avons $a = 1, b = 2$. Posons $\epsilon = \alpha$. Puisque $O(n^{\alpha}) = O(n^{\log_b a + \epsilon})$, nous concluons que $T(n) \in O(n^{\alpha})$.

En choisissant un algorithme efficace pour effectuer le produit de deux grands entiers, de façon à avoir $\alpha = \log_2 3$, on obtient un meilleur temps que l'algorithme itératif, qui lui calcule le $n^{\text{ème}}$ nombre de Fibonacci en $O(n^2)$ (voir BB section 2.7.5).