Introduction to algorithmic Fall 2017

IFT2125-6001 TA: Maëlle Zimmermann

Demonstration 9

1

Question: An n-tally circuit is a circuit that takes n bits as input and produces $1 + \lfloor \log n \rfloor$ bits output. It counts in binary the number of bits equal to 1 in the input. For example, if n = 9 and the input is 011001011, then there are 5 bits equal to 1, and the output is 0101 (5 in binary). An (i, j) -adder is a circuit that takes a number m of i bits and an input number n of bits. It calculates m + n in binary on 1 + max (i, j) bits of exit. For example, if the input is m = 101 and n = 10111 (i = 3, j = 5), the output is the sum of the two numbers, 011100.

It is always possible to build a (i, j) - adder from exactly max (i, j) 3 - tallies. Indeed, to add m + n is to count for each position k the number of bits equal to 1 among the kth bit of m, the kth bit of n, and the eventual bit of detention. As the calculation must be done for max (i, j) k positions we need to max (i, j) 3 - tallies.

1. Use 3-tallies and (i, j) -adders to build an efficient n-tally.
2. Give a recurrence (with initial condition) that describes the number of 3-tallies needed to build the n - tally, including the 3 - tallies that are part of (i, j) - adders.
3. Solve the recurrence exactly.

Solution:

1. We build an n-tally recursively. When $1 \leq n \leq 3$, it suffices to use a 3-tally. When n> 3 we divide the entrance into two by constructing in $\lceil n / 2 \rceil$ - tally and $\lfloor n / 2 \rfloor$ - tally, counting the number of bits equal to 1 in each half of the entrance. The result of these two tallies is summed by a (i, j) - adder where = $1 + \lfloor \log \lceil n / 2 \rceil \rfloor$ and $j = 1 + \lfloor \log \lfloor n / 2 \rfloor \rfloor$.

2. Let t (n) be the number of 3-tallies used to construct an n-tally in the construction given in (1). When $1 \leq n \leq 3$, only one 3-tally is used. When $n > 3$ the number of 3 - tallies used is $t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor)$, plus the number of 3-tallies used to construct the (i, j) -adder, that is max (i, j). As $i = 1 + \lfloor \log \lceil n/2 \rceil \rfloor$ and $j = 1 + \lfloor \log \lfloor n/2 \rfloor \rfloor$, we get

$$t(n) = \begin{cases} 1 & \text{if } 1 \leq n \leq 3, \\ \underbrace{t\left(\left\lceil \frac{\text{not}}{2} \right\rceil\right)}_{\lceil \frac{}{2} \rceil\text{-tally}} + \underbrace{t\left(\left\lfloor \frac{\text{not}}{2} \right\rfloor\right)}_{\lfloor 2 \rfloor\text{-tally}} + \underbrace{1 + \left\lfloor \log \left\lceil \frac{\text{not}}{2} \right\rceil \right\rfloor}_{(i,j)\text{-adder}} & \text{if } n > 3 \end{cases} \tag{1}$$

3. Let $s(i) = t(2^i)$, then we have

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

The characteristic polynomial of the recursion s is $p(x) = (x-2)(x-1)^2$ and so $s(i) = c_1 2^i + c_2 + c_3 i$. Solving the system

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

we get $c_1 = 3$, $c_2 = -2$ and $c_3 = -3$. Thus, $s(i) = 3 \cdot 2^i - 3i - 2$ and therefore $t(n) = s(\log n) = 3n - 3 \log n - 2$ when n is a power of 2.

So we have $t(n) \in \Theta$ (n: n is a power of 2). Since t (n) is possibly not decreasing (we can prove it), we conclude by the rule of harmony that $t(n) \in \Theta(n)$.

Alternatively, if one simply seeks to obtain the order of t and not its form exactly, we can use the theorem seen in class (first case). We have $a = 2$, $b = 2$, and $f(n) = \log(n) \in O(n^{\log 2 - \varepsilon})$ taking any ε it is small (for example 0.1). It is also concluded that $t(n) \in \Theta$ (n: n is a power of 2).

2

Question: Suppose we have access to the following algorithms:

- mult_k1: multiplies a polynomial of degree k with a polynomial of degree 1 in one time O (k),

- mult_kk: multiplies two polynomials of degree k in a time O (k log k).

Let $z_1, ..., z_d \in Z$. Give an efficient algorithm that calculates the unique polynomial p (n) = $a_0 + a_1 n + ... + a_d n^d$ such that ${}_d = 1$ and $p(z_1) = ... = p(z_d) = 0$. Note that we will represent a polynomial $a_0 + a_1 n + ... + a_d n^d$ by the array $[a_0, a_1, ..., a_d]$. Analyze the effectiveness of the algorithm.

Solution: Just calculate the polynomial $p(n) = (n-z_1)(n-z_2) ... (n-z_d)$ recursively by successively cutting the list $z_1, ..., z_d$ in two. Here is such an algorithm:

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

The execution time of zeros is described by the following recursion:

$$t(d) = \begin{cases} 1 & \text{if } d \leq 1, \\ 2t(\lfloor \frac{d}{2} \rfloor) + f(\lfloor \frac{d}{2} \rfloor) & \text{if } d > 1 \text{ and is even,} \\ 2t(\lfloor \frac{d}{2} \rfloor) + t(1) + f(\lfloor \frac{d}{2} \rfloor) + g(d-1) & \text{if } d > 1 \text{ and is odd} \end{cases}$$

o`uf $(d) \in O(d \log d)$ and $g(d) \in O(d)$. So,

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

Let's apply the theorems on recurrences seen in class. We have a = 2, b = 2 and f (d) = dlog d. Let $\varepsilon = 1$. Since $f(d) \in O(d \log d) = O(d^{\log_b a} (\log d)^{\varepsilon})$, we conclude that $t(d) \in O(d^{\log_b a} (\log d)^{\varepsilon + 1}) = O(d (\log d)^2)$.

Question: Let the matrix $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. What happens when we raise A to the power 2? And to the power n? On this principle, build a divide-for-algorithm rule to calculate the `n^{th} element of the Fibonacci sequence. Analyze efficiency of the algorithm with the notation O assuming 1) that the arithmetic operations have a constant cost, then 2) that multiply two integers of size s and q take a time in $\Theta(sq^{\alpha-1})$ if $s \geq q$. Here α is a constant that depends on the algorithm used to make the product. For example, for the efficient algorithm seen in class, we have $\alpha = \log_2 3$. Remember that the size (in bits) of heading `th Fibonacci number is in $\Theta(n)$.

Solution: We notice first that

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

Thus it is sufficient to implement a divide-and-conquer algorithm that calculates the n^{th}`powerful of the matrix A. The method is similar to the expoDC algorithm which calculates the `n^{th} power of a number a BB in Section 7.7. Indeed:

$$A^n = \begin{cases} AT & \text{if } n = 1, \\ (A^{n/2})^2 & \text{if } n > 1 \text{ and is even} \\ AA^{n-1} & \text{if } n > 1 \text{ and is odd} \end{cases}$$

Let T (n) be the recurrence that describes the time of the algorithm, and let M (s, q) be the time for multiply two integers of size s and qo`us $\geq$ q.

If n is even, the matrix to squared is $\begin{pmatrix} F_{n/2-1} & F_{n/2} \\ F_{n/2} & F_{n/2+1} \end{pmatrix}$. It takes 8 multiplication of numbers of maximum size $\frac{n}{2} + 1$. So raise the matrix squared takes a time bounded by $8M(\frac{n}{2} + 1, \frac{n}{2} + 1)$.

If n is odd, you have to perform the product between $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ and $\begin{pmatrix} F_{n-2} & F_{n-1} \\ F_{n-1} & F_n \end{pmatrix}$. it takes 8 multiplications of a number of size 1 with a number of maximum size n. So to perform the product of the two matrices takes a time bounded by 8M (n, 1). We get:

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1, \\ T(\frac{n}{2}) + 8M(\frac{n}{2} + 1, \frac{n}{2} + 1) & \text{if } n > 1 \text{ and is even,} \\ T(n-1) + 8M(n, 1) & \text{if } n > 1 \text{ and is odd} \end{cases}$$

We must therefore find a bound that applies to the even and odd case. Note that if n is odd, we have:

$$T(n) \leq T(n - 1) + 8M(n, 1)$$
$$= T\left(\frac{n - 1}{2}\right) + 8M\left(\frac{n - 1}{2} + 1, \frac{n - 1}{2} + 1\right) + 8M(n, 1))$$

This implies that ∀n> 1,

$$T(n) \leq T\left(\left\lfloor \frac{\text{not}}{2} \right\rfloor\right) + 8M\left(\left\lfloor \frac{\text{not}}{2} \right\rfloor + 1, \left\lfloor \frac{\text{not}}{2} \right\rfloor + 1\right) + 8M(n, 1)).$$

Let's analyze the efficiency of the algorithm according to each assumption.

1) If arithmetic operations and especially multiplications have a cost constant (note that this assumption is unrealistic), we get

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

Thus we obtain by applying the theorem on recurrences seen in class (case 3 with $\varepsilon = 0$) than $T(n) \in O(\log n)$.

2) On the other hand if $M(s, q) \in \Theta(sq^{\alpha-1})$, then $8M(\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 1) \in \Theta((\lfloor \frac{n}{2} \rfloor + 1)(\lfloor \frac{n}{2} \rfloor + 1)^{\alpha-1}) = \Theta((\lfloor \frac{n}{2} \rfloor + 1)^{\alpha}) = \Theta(n^{\alpha})$ and $8M(n, 1) \in \Theta(n1^{\alpha-1}) = \Theta(n)$. As $\alpha > 1$, We have

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

Let's apply the theorem on recurrences seen in class (case 2). We have $a = 1$, $b = 2$. Let $\varepsilon = \alpha$. Since $O(n^{\alpha}) = O(n^{\log_b a + \varepsilon})$, we conclude that $T(n) \in O(n^{\alpha})$.

By choosing an efficient algorithm to perform the product of two large integers, in order to have $\alpha = \log_2 3$, we obtain a better time than the iterative algorithm, which it calculates the n^{th} Fibonacci number in $O(n^2)$ (BB see Section 2.7.5).