Home Assignment 2

Christopher Rydell & Christoffer Falkovén

December 12, 2016

1 Growth of Functions

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

1.1

Prove that $F(n) = O(2^n)$.

We need to prove that $F(n) \leq c_2 2^n$ where $c_2 > 0$ for every value $n > n_0$. We'll begin by assigning the values n_0 and c_2 so that we have a case where we know the condition to be true. For this we'll use $n_0 = 0$ and $c_2 = 1$. Next, we'll prove the inequality using complete induction.

First off we show that the inequality is valid for the base case $n = 0$:

$P(0)$:

$$F(0) = 0 \leq 1 = 1 * 2^0$$

Assume that the following applies for all values of n where $0 \leq n \leq m$:

$P(m)$:

$$F(n) \leq 2^n$$

We'll now show that the inequality also applies for $m + 1$:

$P(m+1)$:

$$\begin{aligned} F(m+1) &= F(m) + F(m-1) \\ &\leq 2^m + 2^{m-1}, \text{ from our assumption} \\ &= 2^m + \frac{2^m}{2} \\ &= 2^m * (1 + \frac{1}{2}) \\ &< 2^m * 2 \\ &= 2^{m+1} \end{aligned}$$

As the base case applies and $P(m) \implies P(m+1)$, we prove that the inequality applies for all $n \in \mathbb{N}$, $n \geq 0$ according to the principle of induction.

□

1.2

Prove that $F(n) = \Omega(2^{\frac{n}{2}})$.

We need to prove that $c_1 2^{\frac{n}{2}} \leq F(n)$ where $c_1 > 0$ for every value $n > n_0$. We'll begin by assigning the values n_0 and c_1 so that we have a case where we know the condition to be true. For this we'll use $n_0 = 2$ and $c_1 = \frac{1}{2}$. Next, we'll prove the inequality using complete induction.

First off we show that the inequality is valid for the base case $n = 2$:

$P(2)$:

$$\frac{1}{2}2^{\frac{2}{2}} = \frac{1}{2}2 = 1 \leq 1 = 1 + 0 = F(1) + F(0) = F(2)$$

Assume that the following applies for all values of n where $2 \leq n \leq m$:

$P(m)$:

$$\frac{1}{2}2^{\frac{n}{2}} \leq F(n)$$

We'll now show that the inequality also applies for $m + 1$:

$P(m + 1)$:

$$\begin{aligned}
\frac{1}{2}2^{\frac{m+1}{2}} &= \frac{1}{2}2^{\frac{m}{2}}(\sqrt{2}) \\
&< \frac{1}{2}2^{\frac{m}{2}}(1 + \frac{1}{\sqrt{2}}) \\
&= \frac{1}{2}2^{\frac{m}{2}} + \frac{1}{2}2^{\frac{m-1}{2}} \\
&\leq F(m) + F(m-1), \text{ from our assumption} \\
&= F(m+1)
\end{aligned}$$

As the base case applies and $P(m) \implies P(m+1)$, we prove that the inequality applies for all $n \in \mathbb{N}$, $n \geq 2$ according to the principle of induction.

□

2 From Code to Recurrences

2.1

```
append [] ys     = ys
append (x:xs) ys = x :  append xs ys
```

If the function is given an empty list for xs, it will simply return ys, and will only take the time for the function to return a list. If xs has elements in it, it will add the element as the head to a list returned by a recursive call to the function, which is given a list with one less element. Therefore the run-time cost can be described as:

$$T_{append}(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T_{append}(n-1) + t_{add} & \text{if } n > 0 \end{cases}$$

2.2

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

When n is either 0 or 1 the function will simply take the time for returning a value. If n > 0 the function will make two recursive calls to itself, one where n is decreased by 1 and one where it is decreased by 2. This will also add the time it takes to add the returned values of the recursive calls, which makes the run-time cost of the function:

$$T_{fib}(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T_{fib}(n-1) + T_{fib}(n-2) + t_{sum} & \text{if } n > 0 \end{cases}$$

2.3

```
power b 0 = 1
power b n | even n = power b (n ‘div‘ 2) * power b (n ‘div‘ 2)
power b n | odd n = b * power b (n ‘div‘ 2) * power b (n ‘div‘ 2)
```

When n = 0 the function simply takes the time to return a value. If it's larger, it checks if n is even or odd, which adds time, and depending on the result also calls the multiplication function. As all three of the functions even, odd and (*) have a constant time complexity, they can be described as $\Theta(1)$ in the run-time cost. The function also makes two recursive calls to itself, with the value n divided by 2. The run-time cost is as follows:

$$T_{power}(n) = \begin{cases} t_0 & \text{if } n = 0 \\ \Theta(1) + 2T_{power}(n/2) & \text{if } n > 0 \end{cases}$$

3 Solving Recurrences

3.1

$$f(n) := \begin{cases} 1 & \text{if } n = 0 \\ 3 & \text{if } n = 1 \\ 2f(n-1) + 3f(n-2) & \text{if } n > 1 \end{cases}$$

We'll attempt to find a pattern in the recurrence using the expansion method.

$$\begin{aligned} f(0) &= & 1 \\ f(1) &= & 3 \\ f(2) &= 2f(1) + 3f(0) = 6 + 3 = & 9 \\ f(3) &= 2f(2) + 3f(1) = 18 + 9 = & 27 \\ f(4) &= 2f(3) + 3f(2) = 54 + 27 = & 81 \end{aligned}$$

We see that the following pattern emerges from our analysis:

$$f(n) = 3^n, \quad n \in \mathbb{N}$$

3.2

$$g(n) := \begin{cases} 0 & \text{if } n = 0 \\ g(n-1) + 2n - 1 & \text{if } n > 0 \end{cases}$$

We look at the function using the substitution method.

$$\begin{aligned} g(n) &= g(n-1) + 2n - 1 \\ &= (g(n-2) + 2(n-1) - 1) + 2n - 1 \\ &= g(n-2) + 4n - 4 \\ &= (g(n-3) + 2(n-2) - 1) + 4n - 4 \\ &= g(n-3) + 6n - 9 \\ &= (g(n-4) + 2(n-3) - 1) + 6n - 9 \\ &= g(n-4) + 8n - 16 \\ &= g(n-k) + 2kn - k^2 \\ &\vdots \\ &= g(n-n) + 2n^2 - n^2 \quad \text{when } k = n \\ &= 2n^2 - n^2 \\ &= n^2 \end{aligned}$$

As can clearly be seen from the substitution, the function can be rewritten to a closed form as:

$$g(n) = n^2, \quad n \in \mathbb{N}$$

4 The Master Theorem

$$T(n) := 3 * T(n/2) + n^2 * \log(\log n)$$

We will check if the function behaves according to any of the master theorem cases, starting with the first one.

4.1 Case 1

For the first case we need to check if $f(n) = O(n^c)$, where $\log_b a < c$, in which $f(n) = n^2 \log(\log n)$, $b = 2$ and $a = 3$ based on the given function. As $2^2 = 4$, we can safely set $c = 2$ as it is definitely greater than $\log_2 3$. As $\log(\log n)$ will eventually reach a value greater than 1 and then increase from there, we can determine that $f(n) \neq O(n^c)$ and move on to the second case.

4.2 Case 2

For the second case we check to see if $f(n) = \Theta(n^c \log^k n)$, where $c = \log_b a$ and $k \geq 0$. We can set the value $k = 2$ and immediately tell that $n^2 \log(\log n)$ will be growing at a greater rate than $n^{\log_2 3} \log^2 n$, which makes the second case invalid.

4.3 Case 3

For the third case we look if $f(n) = \Omega(n^c)$, where $c > \log_b a$. We'll set the value c to any arbitrary number $\log_2 3 < c < 2$ and see that n^c will in fact grow at a slower rate than $n^2 \log(\log n)$, fulfilling the case.

4.4 Regularity condition

We also need to make sure that the regularity condition holds for the master theorem to be applicable. The regularity condition states that $af(\frac{n}{b}) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n, so we insert in our function to see if it works.

$$\begin{aligned}
af(\frac{n}{b}) &\leq kf(n) \\
3f(\frac{n}{2}) &\leq kf(n) \\
3(\frac{n}{2})^2 \log(\log \frac{n}{2}) &\leq kn^2 \log(\log n) \\
\frac{3}{4}n^2 \log(\log \frac{n}{2}) &\leq kn^2 \log(\log n)
\end{aligned}$$

We know that $\log(\log \frac{n}{2}) < \log(\log n)$ and can thereby safely say that the inequality applies for any $\frac{3}{4} \leq k < 1$, which proves that the regularity condition holds. According to the master theorem, we can now determine that $T(n) = \Theta(n^2 * \log(\log n))$.

5 Example: Quicksort

5.1

When the function is called with the empty list `xs`, the run-time will be the one it takes to simply return a tuple of two empty lists. If the list isn't empty, it makes a single recursive call to itself with a list one element shorter and adds the time it takes to add an element at the head of a list. The run-time cost can thus be described with the following recurrence:

$$T_{partition}(n) := \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_{add} & \text{if } n > 0 \end{cases}$$

5.2

We'll use the expansion method to find a pattern for $T_p(n)$ and determine a closed function.

$$\begin{aligned} T_p(0) & & &= t_0 \\ T_p(1) &= T_p(0) + t_{add} & &= t_0 + t_{add} \\ T_p(2) &= T_p(1) + t_{add} & &= t_0 + 2t_{add} \\ T_p(3) &= T_p(2) + t_{add} & &= t_0 + 3t_{add} \\ &\vdots \\ T_p(n) & & &= t_0 + nt_{add} \end{aligned}$$

5.3

(a) `lows == highs`
If the list `xs` given to the function is empty, it will take the time it takes to return an empty list. Otherwise, the function will call the partition function with a list 1 element shorter than `xs`. It will also make two recursive calls to itself, each time with a list half the length of `xs` minus one element. There will also be an added cost of using `(:)` and `(++)`, which can be described as t_{add} and $\Theta(n)$ respectively. The recurrence is as follows:

$$T_q(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T_p(n-1) + 2T_q(\frac{n-1}{2}) + \Theta(n) + t_{add} & \text{if } n > 0 \end{cases}$$

(b) `length highs == 0`
In this case the function will make one recursive call to itself with a list with the length of xs minus one (`lows`) and one with the length 0 (`lows`), the latter simply adding t_0 from the base case. The recurrence for the run-time cost is thereby:

$$T_q(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T_p(n-1) + T_q(n-1) + \Theta(n) + t_{add} + t_0 & \text{if } n > 0 \end{cases}$$