Solution Manual For Algorithms by Das Gupta Papadimitriou and Vazirani

Sidharth Arya

October 5, 2018

Contents

Prologue

0.1

- f = Θ(g)

Since `n-100` and `n-200` have the same power of n.

- f = O(g)

Since $n^{1/2}$ is smaller than $n^{2/3}$

- f = Θ(g)

Since `log n` can always be overcome by `n` above a particular `n`, and so can it be less than `log n` below a particular `n`.

- f = Θ(g)

Since 10n log 10n = 10n(log 10 + log n) = 10n log 10 + 10n log n = θ (nlogn)

- f = Θ(g)

Since log 2n = log 2 + log n and log 3n = log 3 + log n. which makes log 2 and log 3, just constants.

- f = Θ (g)

Since log n^2 = 2log n. and 2 is just a constant that can be dropped.

- f = Ω(g)

Since power greater than 0 can always overtake log at some point

- f = Ω(g)

On comparing n^2 is greater than n, and log's are not that signifincant in comparison.

- f = Ω(g)

Since power greater than 0 can always overtake log at some point

- f = Ω(g)

Since (log n)$^{(\log n) -1}$ is greater than n for some value of n. Let us simplify equation for better comparison. We can multiply the equations by log n

Giving

f'(x) = (log n)$^{(\log n) + 1}$

g'(x) = n

I am using ' since equations have been altered and not the same as they initially were, yet they are still comparable.

now applying log

f"(x) = (1+log n) log (log n)

g"(x) = log n

It is clear that it may not be possible to draw a direct relation. But f"(x) can be written as:

f"(x) = (1 + g"(x)) log (g"(x))

Now, if we visualize, log g"(x) will produce 1 at some point and will keep increasing after that value, and at that point it is multiplied by 1+ g"(x) which will clearly compound to something greater than g"(x) itself.

Therefore, at some point f(x) will overtake g(x)

- f = Ω(g)

Since power greater than 0 can always overtake log at some point

- f = O(g)

$5^{\log_2 n} = 5^{\log_2 n \times \log_5 5} = 5^{\log_5 n \times \log_2 5} = (5^{\log_5 n})^{\log_2 5} = n^{\log_2 5} \approx n^{2.\text{something}}$

Therefore, at some n g will overcome f, by comparison of powers.

- f = Ω(g)

Since n2^n will produce a greater value for n than 3^n at some point.

- f = Θ(g)

Since the two differ by a constant multiplicative factor, i.e. 2.

- f = Θ(g)

Expansion of n!

n! = n(n-1)(n-2)..1

Notice that largest possible power of n will be n. if we count unity as (n-something)

Therefore n! can be dependent on n^n

and so n^n at some point will definetely overcome 2^n.

n! grows atleast as fast as 2^n, maybe more, but our concern is satisfied.

- f=O(g)

Taking log on both equations, and comparing

f'(x) = log n x log(log n)

g'(x) = $(\log_2 n)^2$ x log 2 = log 2 x $(\log_2 e)^2$ x log n x log n

Comparing the n-dependent terms, log n is certainly greater than log(log n). Therefore, g'(x) will be greater than f'(x) at some point.

0.2

$g(n) = 1 + c + c^2 \ldots + c^n$

- if $c < 1$

The series will be decreasing. so the maximum number will be 1. Therefore, g(n) be independent of n, on approximation, implying $g(n) = \Theta(1)$.

- if $c = 1$

$g(n) = 1 + 1 + 1 + \ldots + 1 = n$
Therefore, $g(n) = \Theta(n)$

- if $c > 1$

g(n) will be increasing. The largest term will be c^n
Therefore, $g(n) = \Theta(c^n)$

0.3

-

$F_n \geq 2^{0.5n}$ for $n \geq 6$

Induction

Case : n = 6 (Base Case)

$F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$

L.H.S : $F_6 = F_5 + F_4 = 8$

R.H.S : $2^{0.5 \text{ x } 6} = 8$

Since L.H.S = R.H.S

This case is true.

Let it be true, for n = k.

$F_k = F_{k-1} + F_{k-2} \geq 2^{0.5 \text{ x } k}$

Case: n = k+1

$F_{k+1} = F_k + F_{k-1} \geq 2^{0.5 \text{ x } k} + 2^{0.5 \text{ x } k-1} = 2^{0.5\ k}\ (1+2^{-0.5})$

It can easily be shown that $1 + 2^{-0.5} > 2^{0.5}$

Subsituting that instead $F_{k+1} \geq 2^{0.5(k+1)}$

-

We need to solve for c such that:

$F_n \leq 2^{cm} \; \forall \; n \geq 6$
$\Rightarrow F_{n-1} + F_{n-2} \leq 2^{cn}$
$\Rightarrow 2^{c(n-1)} + 2^{c(n-2)} \leq 2^{cn}$
$\Rightarrow 2^{c(n-2)} \; (2^c + 1) \leq 2^{cn}$
$\Rightarrow 2^c + 1 \leq 2^{2c}$
Let $2^c = x$
$\Rightarrow x + 1 \leq x^2$
$\Rightarrow x^2 - x - 1 \geq 0$
$2^c \geq \frac{1 \pm \sqrt{5}}{2}$
Taking log
$\Rightarrow c \geq \log_2 (1 \pm \sqrt{5}) - 1$
$c \approx 1.694$

-

From previous example . $c = 1.694$

0.4

- Basic Matrix multiplication
- If we attribute all multiplicative and additive operations as O(1), then, If we have to compute fibonnacci of, say 9. We already know the value of matrix X. So to find value of fibonnacci 9, we need to know X^9, which can be written as X^8 x X. Since we already know the value of X. All we have to do is compute X^2 = X x X and X^4 = X*2 x X^2 and X^8 = X^4 x X^4, which gives us a total of 3 matrix multiplication operations. for n = 9.

And for n = 8 it would be the same. Since log 8 is 3, so there are 3 matrix multiplication. which concludes the answer.

- It's an obvious statement, since even the final answer will not be n bits long.

let's say we start with 0, 1, 1. The third fibonnacci number in binary will be 10. compare them with n= 0, 1,2,3. Just for sake of argument let's use induction.
Induction

for n = 1 (Base Case)
$F_1 = 1$
also in binary only one bit is needed. Base Case is satisfied.
let's assume it to be true for n=k
then
$F_{k+1} = F_k + F_{k-1}$
We know that binary F_k = O(k) and binary F_{k-1} = O(k-1)

It is already known that the maximum of n digit number on addition with itself will have n+1 digits.

e.g
1111111 + 1111111 = 11111110
Therefore k-digit binary number + k-1 digit binary number cannot produce more digits than k+1...
Hence the statement is true for all fibonnacci numbers.

- Since log n matrix multiplications are required, each compounding to 8 matrix multiplications. Running time will be nearly O(8M(n) log n) = O(M(n) log n)
- Basically as we raise the power of the matrices, the length increases (doubles). Therefore

M(1) + M(2) + M(4)... + M(n-1) = O(M(n)) Since the terms have some dependence on n (e.g. M(n-1)).

0.5 Extra

0.5.1 Fibonnaci 1: By Recursion

```
#include<stdio.h>

int fib(int n)
{
  if(n == 0) return 0 ;
  if(n == 1) return 1;
  else return fib(n-1) + fib(n-2);
}
int main()
{
  printf("%d", fib(40));
   return 0;
}
```

Time: 0m1.143s Note: if you have a linux machine, you can check the time on your machine by `time output-file`. but the time will be different on your machine.

0.5.2 Fibonnaci 1: By Loop

```
#include<stdio.h>

int fib(int n)
{
  if(n == 0) return 0 ;
  if(n == 1) return 1;
  int Arr[n];
```

```
  Arr[0] = 0; Arr[1] = 1;
  for(int i = 2 ; i < n; i++)
    Arr[i] = Arr[i-1] + Arr[i-2];
  return Arr[n-1] + Arr[n-2];
}
int main()
{
  printf("%d", fib(40));
   return 0;
}
```

Time: 0m0.002s

0.5.3 Fibonnaci 1: By Matrices

```
#include<stdio.h>

int fib(int n)
{
  if(n == 0) return 0 ;
  if(n == 1) return 1;
  int mat[2][2] = {
    {0, 1},
    {1, 1}
}

}
int main()
{
  printf("%d", fib(40));
   return 0;
}
```

Time: 0m0.002s

Chapter 1

Algorithms with numbers

Code

```
(add-to-list 'org-latex-classes
             '("some"
               "\\documentclass{book}"
               ("\\chapter{%s}" . "\\chapter*{%s}")
               ("\\section{%s}" . "\\section*{%s}")
               ("\\subsection{%s}" . "\\subsection*{%s}")
               ("\\subsubsection{%s}" . "\\subsubsection*{%s}")))
```