Data Structures and Algorithms

Spring 2014

Cesar Agustin Garcia Vazquez

February 7, 2014

1 Problem Set 01

1. Suppose you are given a six-sided die, that might be biased in an unknown way. Explain how to use die rolls to generate unbiased coin flips, and determine the expected number of die rolls until a coin flip is generated. Now supposed you want to generate unbiased die rolls (from a six-sided die) given your potentially biased die. Explain how to do this, and again determine the expected number of biased die rolls until an unbiased die roll is generated. For both problems you need not give the most efficient solution; however, your solution should be reasonable, and exceptional solutions will receive exceptional scores.

   Answer: To solve this problem, we can start with a small generalization of the problem if we can see a coin as a 2 sided die. So assuming that we have an n-sided die, and each side has its corresponding probability $p_1, p_2, \ldots, p_n$. We can throw the die n times and if no repetition occurs, then we have a valid die roll. The number of different sequences that can be generated by throwing a die is n^n, i.e., in each turn of throw any of the n sides can come up (maybe one side with more probability than other, but at the end there are n possible outcomes).
   The number of valid sequences that generate a fair die roll is $n!$, so if we consider $p_1 \cdot p_2 \cdot \ldots \cdot p_n = \mathcal{P}$, then $n! \cdot \mathcal{P}$ is the probability of generating a valid fair die roll and $1 - n! \cdot \mathcal{P}$ is the probability of repeating the event since the sequence is invalid. Based on all this, we can get the general equation (1) for

the expected number of throws of a biased n-sided die.

$$\begin{aligned}
T &= n + (1 - n! \cdot \mathcal{P}) \cdot T \\
T - (1 - n! \cdot \mathcal{P})T &= n \\
T[1 - (1 - n! \cdot \mathcal{P})] &= n \\
T(1 - 1 + n! \cdot \mathcal{P}) &= n \\
T(n! \cdot \mathcal{P}) &= n \\
T &= \frac{n}{n! \cdot \mathcal{P}} \\
T &= \frac{1}{(n-1)! \cdot \mathcal{P}} \qquad (1)
\end{aligned}$$

To determine the sequence that correspond to each side i, we just match the first element of the sequence with the side of the die. For instance, if we have the sequence $[1, 6, 8, 1, 5, \ldots, i_n]$, then the sequence yields a die roll with side i being up.

Now, for the case $n = 6$ (a six-sided die), the expected number of throws before stoping and considering $\mathcal{P} = \prod p_i$, where each p_i represents the probability of each side i, where $i \in \{1, 2, 3, 4, 5, 6\}$, is shown in equation(2).

$$\begin{aligned}
T &= \frac{1}{(n-1)! \cdot \mathcal{P}} \\
&= \frac{1}{(6-1)! \cdot \mathcal{P}} \\
&= \frac{1}{5! \cdot \mathcal{P}} \\
&= \frac{1}{120 \cdot \mathcal{P}} \qquad (2)
\end{aligned}$$

To use the biased die to generate coin flips we just need to get a sequence that generates a unbiased die roll and use the rule shown in table 1. For instance, if we get the sequence $s_0 = 1, 6, 5, 3, 2, 4$, then s_0 represents a thrown die with the face of the number 1 up. Since 1 is odd, then we have tails.

The expected number of die rolls to generate a flip is the same as the expected number of die rolls to generate an unbiased roll.

Table 1: Rule to determine Heads or Tails based using a biased die

Flip	Rule
Head	If the sequence generates an even side
Tail	If the sequence generates and odd side

2. On a platform of your choice, implement the three different methods for computing the Fibonacci numbers (recursive, iterative, and matrix) discussed in lecture. Use integer variables. How fast does each method appear to be? Give precise timings if possible. (This is deliberately open-ended; give what you feel is a reasonable answer. You will need to figure out how to time processes on the system you are using, if you do not already know.) Can you determine the first Fibonacci number where you reach integer overflow? (If your platform does not have integer overflow -lucky you! - you might see how far each process gets after five minutes.)
   Since you should reach integer overflow with the faster methods quite quickly, modify your programs so that they return the Fibonacci numbers modulo $65536 = 2^{16}$. (In other words, make all of your arithmetic module 2^{16} - this will avoid overflow! You must do this regardless of whether or not your system overflows.) For each method, what is the largest Fibonacci number you can compute in one minute of machine time?

   Answer: The tables 2a and 2b present the results of calculating n first Fibonacci numbers, using the iterative and matrix methods. From those results the iterative method appears to be much faster than the matrix method. The results were obtained using a MacBook with a 2 GHz Intel Core 2 Duo and 8 GB in RAM and the methods were programmed in Java.
   If we take a look at table 2a, we can notice that it is hard to make an assumption about the running time while calculating less than one million Fibonacci numbers. After that threshold is reached, the behavior shown looks like linear time if we consider $100,000,000$ hour based number, every $100,000,00$ added to it, adds around 400 milliseconds. From here, if we add the time it took to compute $400,000,000$ and $800,000,000$ Fibonacci numbers, we get $1,652$ and the time it took to compute $400,000,000 + 800,000,000 = 1,200,000,000$ was $1,654$, which means that our assertion about linear time holds.

   For the results in table 2b, also the threshold of one million has to be passed in order to obtain values from which we can deduct something. If we only

take a look at table 2b and forget about the results in table 2a, we might infer that algorithm is running in linear time. If we take the running time of $100,000,000$ and multiply it by 2, we get $329,190$ and the running time of calculating $200,000,000$ Fibonacci numbers is $349,489$, i.e., just a little higher than we expected. If we add the running time of $100,000,000$ and $200,000,000$, we get $514,084$, which is just a little lower than the actual output $527,118$.
The linear time starts to mismatch while calculating $400,000,000$ Fibonacci numbers, for which we can consider multiplying by 2 the running time of calculating $200,000,000$ Fibonacci numbers, which outputs $698,978$ or we can also add the running time of calculating $300,000,000$ and $100,000,000$ Fibonacci numbers, which outputs $691,713$. For any of those two cases, the expected results are starting to be very low compared to the actual result $757,039$.
But if we multiply by 2 the running time of computing $400,000,000$ Fibonacci numbers, we get $1,514,078$ which is very close to $1,527,136$. Also if we add the running time of computing $400,000,000$ and $800,000,000$ we get $2,284,175$ which is very close to $2,276,780$. So based on all this, it might appear that the matrix method also runs in linear time but with a much bigger constant c.

The table 2 presents the results of using the typical recursive method. This method appeared to be extremely slow even for small instances of the problem. In no case a linear pattern appears, but if we assume that it has an exponential pattern then using logarithms we can compute that the base is around 1.25.

To determine the first Fibonacci number where I reach integer overflow, first I get the maximum number that the language supports as an integer and then I add 1. In Java the maximum integer value is $2,147,483,647$ and when 1 is added, I get the value $-2,147,483,648$, so no overflow happens per se, but the value is incorrect. So to detect overflows, I just need to detect if the number change from positive to negative. For this part of the exercise, I will use the iterative method since it seems to be the fastest of the three. To determine the number where the overflow occurs, I will use binary search. I will start with 2 and then multiply by 2 until I reach a changed of sign.
Based on the above I get that the $F_{46} = 1,836,311,903$ and $F_{47} = -1,323,752,223$. So $n = 47$ is when the first overflow occurs.

Using arithmetic modulo $65,536$, in one minute the iterative method was able to calculate $96,533$ Fibonacci numbers and the recursive method was able to calculate only 46. Compared to tables 2a performed really bad. One reason is that the iterative method reaches overflow so fast that the module operation

Figure 1: Tables of results from methods that ended relatively fast

Fibonacci numbers calculated	Milliseconds
10	0
20	0
40	0
80	0
160	1
320	2
640	0
1,280	1
2,560	4
5,120	17
10,240	42
20,480	0
40,960	0
81,920	0
819,200	1
8,192,000	11
81,920,000	144
100,000,000	213
200,000,000	323
300,000,000	407
400,000,000	559
800,000,000	1,093
1,200,000,000	1,654

(a) Fibonacci numbers and time using Iterative method

Fibonacci numbers calculated	Milliseconds
10	0
20	0
40	0
80	1
160	3
320	6
640	32
1,280	9
2,560	28
5,120	32
10,240	39
20,480	34
40,960	41
81,920	132
819,200	1,505
8,192,000	11,406
81,920,000	127,880
100,000,000	164,595
200,000,000	349,489
300,000,000	527,118
400,000,000	757,039
800,000,000	1,527,136
1,200,000,000	2,276,780

(b) Fibonacci numbers and time using Matrix method

is applied much more times than in the iterative method, i.e., if we consider $F_{23} = 28,657$ and $F_{24} = 46,368$, the sum is $F_{25} = 75,025\%65,536 = 9,489$, then for $F_{26} = 9,489+46,368 = 55,857$, $F_{27} = 55,857+9,489 = 65,346$, which implies that for $F_{28} = 65,346 + 55,857 = 121,203\%65,536 = 55,667$.
For the matrix method, the maximum number of Fibonacci numbers that it was able to compute in one minute was $9,149,862$, which makes sense according to table 2b, due to the modulo operations, but this time it was so much faster than the iterative method. In fact, to verify that the computation was correct, I took the time to compute the $96,533$-th Fibonacci number using arithmetic

Table 2: Fibonacci numbers and time using Recursive method

Fibonacci numbers calculated	Milliseconds
10	0
20	5
30	23
32	72
33	115
34	160
35	249
36	398
37	657
38	1,044
39	1,699
40	2,682
41	4,531
42	7,280
43	11,798
44	19,487
45	30,321
46	48686
50	334,363

modulo 65,536 and the iterative method took 7 milliseconds and the matrix method took only 2 milliseconds. Again, even though we also apply the modulo operation, for the matrix case we use the function only $O(lg(n))$ times and for the iterative method, we use it O(n) as n goes bigger.

3. Indicate for each pair of expressions (A, B) in the table below the relationship between A and B. Your answer should be in the form of a table with a "yes" or "no" written in each box. For example, if A is $O(B)$, then you should put

a "yes" in the first box.

A	B	O	o	Ω	ω	Θ
$\log n$	$\log(n^2)$	Yes	No	Yes	No	Yes
$\log(n!)$	$\log(n^n)$	Yes	No	Yes	No	Yes
$\sqrt[3]{n}$	$(\log n)^6$	No	No	Yes	Yes	No
$n^2 2^n$	3^n	Yes	Yes	No	No	No
$(n^2)!$	n^n	No	No	Yes	Yes	No
$\frac{n^2}{\log n}$	$n \log(n^2)$	No	No	Yes	Yes	No
$(\log n)^{\log n}$	$\frac{n}{\log(n)}$	No	No	Yes	Yes	No
$100n + \log n$	$(\log n)^3 + n$	Yes	No	Yes	No	Yes

4. For all of the problems below, when asked to give an example, you should give a function mapping positive integers to positive integer. (No cheating with 0's!)

   a) Find (with proof) a function f_1 such that $f_1(2n)$ is $O(f_1(n))$.

   Answer: Consider the function $f_1(n) = n$ and $c = 2$, then

   $$f_1(2n) = 2n \leq c \cdot n = 2n \tag{3}$$

   Based on the equation (3), there exists a constant $c = 2$ such that $f_1(2n) \leq cf_1(n)$, which means that $f_1(2n)$ is $O(f_1(n))$. To prove that is valid for $n \geq 1$, we begin with the base case $n = 1$, which yields the inequality (4).

   $$\begin{aligned} f_1(2n) &= f_1(2 \cdot 1) \\ &= f_2(2) \\ &= (2) \\ &= 2 \\ &= 2f_1(1) \end{aligned} \tag{4}$$

   Assuming that the claim is true for n, then $f_1(2n) \leq cf_1(n)$. For the induction step, considering $n+1$, we have the inequality (5), which finished the proof.

$$\begin{aligned} f_1(2(n+1) &= f_1(2n+2) \\ &= (2n+2) \\ &= 2n+2 \\ &= 2(n+1) \\ &= 2f_1(n+1) \qquad (5) \end{aligned}$$

b) Find (with proof) a function f_2 such that $f_2(2n)$ is not $O(f_2(n))$.

Answer: The problem is asking to find a non-2-smooth function. Considering $f_2(n) = 2^n$, then we have the result in equation (6).

$$\begin{aligned} f_2(2 \cdot n) &= 2^{2\cdot n} \\ &= (2^2)^n \\ &= 4^n \qquad (6) \end{aligned}$$

Using limits, as in equation (7), we can see that 2^n is $o(4^n)$, which means 4^n is $\omega(2^n)$, i.e., $f_2(2n)$ is $\omega(f_2(n))$.

$$\begin{aligned} \lim_{x\to+\infty} \frac{2^n}{4^n} &= \lim_{x\to+\infty} \left(\frac{2}{4}\right)^n \\ &= \lim_{x\to+\infty} \left(\frac{1}{2}\right)^n \\ &= \lim_{x\to+\infty} \frac{1}{2^n} \\ &= 0 \qquad (7) \end{aligned}$$

Another way to proof that $f_2(n) = 2^n$ is a non-2-smooth function is by contradiction. Assume that there exists a constant c such that $4^n \leq c \cdot 2^n$, then we reach the inequality (8), which is a contradiction as n keeps growing.

$$\begin{aligned} 4^n &\leq c \cdot 2^n \\ \frac{4^n}{2^n} &\leq c \\ \left(\frac{4}{2}\right)^n &\leq c \\ 2^n &\leq c \qquad (8) \end{aligned}$$

c) Prove that if $f(n)$ is $O(g(n))$, and $g(n))$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Answer: If $f(n)$ is $O(g(n))$, and $g(n))$ is $O(h(n))$, then the inequality (9) holds for a constant $c_1 \in \mathbb{R}+$ and $n \geq n_0$ with $n_0 \in \mathbb{N}$.

$$f(n) \leq c_1 \cdot g(n) \tag{9}$$

And also, the inequality (10) holds for a constant $c_2 \in \mathbb{R}+$ and $n \geq n_1$, with $n_1 \in \mathbb{N}$.

$$g(n) \leq c_2 \cdot h(n) \tag{10}$$

If we take the inequality $g(n) \leq c_2 \cdot h(n)$ and multiply by c_1 on both sides, then we have the inequality $c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$, and if we consider $c' = c_1 \cdot c_2 \in \mathbb{R}+$, then by inequality (9), we have the inequality (11) that holds by taking the greater between n_0 and n_1, otherwise, there might be a finite number of values ($|n_0 - n_1|$), for which the inequality might not hold.

$$f(n) \leq c_1 \cdot g(n) \leq c'h(n), \forall n \in \mathbb{N} : n \geq \max\{n_0, n_1\} \tag{11}$$

Based on the inequality (11) and the transitivity of $\leq$, we have that $f(n) \leq c'h(n)$, which means $f(n)$ is $O(h(n))$.

d) Give a proof or a counterexample: if f is not $O(g)$, then g is $O(f)$.

Answer. The problem is asking if the ordering on functions induced by the O notation is total. By considering the functions f and g as defined in equations (12) and (13) respectively, we have that $f(n)$ is not $O(g(n)$ and $g(n)$ is not $O(f(n))$.

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ n^2 & \text{if } n \text{ is odd} \end{cases} \tag{12}$$

$$g(n) = \begin{cases} n^3 & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases} \tag{13}$$

The proof is by contradiction. Assume that $f(n)$ is $O(g(n))$, i.e., $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that $\forall n \in \mathbb{N}$ and $n \geq n_0$, we have that $f(n) \leq c \cdot g(n)$. In this case, n_0 has to be even so $f(n) = 1 \leq g(n) = n^3$, but $n_0 + 1$ is odd, so $f(n_0 + 1) = (n_0 + 1)^2 \geq 1$, because the minimum positive even number is 2, so $n_0 + 1 \geq 3 \geq 1$. So we have a contradiction. If we had started considering n_0 as an odd number, then for $n_0 = 1$ we have that

$f(n_0) = (1)^2 = 1 = g(n_0)$, which makes it true, as well as for $n_0 = 2$, but then for $n_0 = 3$ we have $f(3) = (3)^2 = 9 < 1 = g(3)$, which is a contradiction, so $f(n)$ is not $O(g(n))$.
If we assume that $g(n)$ is $O(f(n))$, then $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that $\forall n \in \mathbb{N}$ and $n \geq n_0$, we have that $g(n) \leq c \cdot f(n)$. In this case, n_0 has to be odd so $g(n) = 1 \leq g(n) = n^2$, but $n_0 + 1$ is even, so $g(n_0 + 1) = (n_0 + 1)^3 > f(n_0 + 1) = 1$, which contradicts the fact that the statement is true for all $n \geq n_0$. If we had started considering n_0 as an even number, then we have that $g(n_0) = n_0^3 > f(n_0) = 1, \forall n \geq 2$, which is a contradiction, so $g(n)$ is not $O(f(n))$.

e) Give a proof or a counterexample: if f is $o(g)$, then f is $O(g)$.

Answer: One of the easiest way of proving this fact is by using the definition of o, which means that $\exists c, n$ such that the inequality (14) holds.

$$\begin{aligned} f(n) &< c \cdot g(n) \\ &\leq c \cdot g(n) \end{aligned} \tag{14}$$