CSOR W4231.001 – Spring, 2023

Brief solutions to homework 1

1. Solution to problem 1

   (a) Since $\log N! = \log 1 + \log 2 + \ldots + \log N$, and each of the N terms has at most n bits, $N!$ is $\Theta(N \cdot n)$ bits long (the upper bound is obvious; a matching lower bound is easy to show as well—*how?*).

   (b) We can compute $N!$ as follows:

Algorithm 1

`Factorial`(N) //N should be a positive integer

```
1: r = 1
2: for i = 2 to N do
3:     r = r · i
4: end for
5: return r
```

Running time: N iterations, each multiplying two numbers consisting of at most $N \cdot n$ bits. Hence the time is $O(N \cdot (Nn)^2) = O(N^3 n^2)$.

2. Solution to problem 2

   (a) i. If a permutation is missing, and the sorted input corresponds to that permutation, then the algorithm will not sort correctly when provided with that input.

   ii. Thus the tree has at least $n!$ leaves.

   iii. It corresponds to the length of the longest root-to-leaf path in the tree. The latter quantity is also called the depth of the tree.

   iv. A binary tree of depth d has at most $2^{d+1} - 1$ nodes, and at most 2^d leaves. Equality is achieved by the complete binary tree of depth d, where every level of the tree *including the last* is completely filled. (The proof is by straightforward induction.) This also implies that a binary tree with x leaves has depth at least $\log_2 x$.

   v. Since the tree corresponding to the algorithm must have $n!$ leaves, the tree has depth at least $\log n! = \Omega(n \log n)$. It follows that the worst-case time complexity is $\Omega(n \log n)$.

   (b) Initialize $D + 1$ counters to 0, where we maintain one counter for each of the possible values of the array items. The goal is to store the number of items of each value in the counters. We can do so by a simple pass over the input in time $O(n)$. We can then go over the counters in ascending order and produce the sorted output in $O(n + D)$ time.

   (c) No, because it is not comparison-based.

3. **Solution to problem 3**

Split the n-dimensional vector $\mathbf{v}$ into vectors $\mathbf{v}^t$ and $\mathbf{v}^b$ which consist of the top and bottom $n/2$ entries of $\mathbf{v}$, respectively.

Similarly, split the n-dimensional vector $H_k\mathbf{v}$ into $(H_k\mathbf{v})^t$ and $(H_k\mathbf{v})^b$. Then

$$\begin{aligned}(H_k\mathbf{v})^t &= H_{k-1}\mathbf{v}^t + H_{k-1}\mathbf{v}^b = H_{k-1}(\mathbf{v}^t + \mathbf{v}^b)\\ (H_k\mathbf{v})^b &= H_{k-1}\mathbf{v}^t - H_{k-1}\mathbf{v}^b = H_{k-1}(\mathbf{v}^t - \mathbf{v}^b)\end{aligned}$$

To compute $H_k\mathbf{v}$ first compute $\mathbf{v}^t+\mathbf{v}^b$ and $\mathbf{v}^t-\mathbf{v}^b$ and then compute the products $H_{k-1}(\mathbf{v}^t+\mathbf{v}^b)$ and $H_{k-1}(\mathbf{v}^t-\mathbf{v}^b)$ recursively.

Running time: Let $T(n)$ be the time to multiply the $n\times n$ matrix H_k by an n-dimensional vector $\mathbf{v}$. Then $T(n) = cn + 2T(n/2) = O(n\log n)$.

4. **Solution to problem 4**

(a) We will use strong induction on $n \geq 6$. Note that $F_6 = 8, F_7 = 13$.

Base case: For $n = 6$, $n = 7$, we have $F_6 = 8 \geq 2^3$, $F_7 \geq 2^3.5$.

Hypothesis: Assume that $F_n \geq 2^{m/2}$, for $6 \leq m < n$.

Step: We will show that $F_n \geq 2^{n/2}$. By the definition of the n-th Fibonacci number and the inductive hypothesis, we have

$$F_n = F_{n-1} + F_{n-2} \geq \ 2\cdot F_{n-2} \geq 2\cdot 2^{\frac{n-2}{2}} = 2^{\frac{n}{2}}$$

(b) Algorithms `fib`(n) and `iter-fib`(n) at the end of the solution are recursive and iterative algorithms for computing F_n, respectively.

i. Running time of `fib`(n): $\quad T(n) = T(n-1) + T(n-2) + O(1) \geq F_n = \Omega(2^{n/2})$.

ii. Running time of `iter-fib`(n): $\quad O(n) \quad$ (n iterations, each takes $O(1)$ time)

iii. We will now discuss a matrix solution. Let $M = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$. Then

$$\begin{bmatrix}F_n\\ F_{n-1}\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-1} \cdot \begin{bmatrix}F_1\\ F_0\end{bmatrix} = M^{n-1}\cdot\begin{bmatrix}F_1\\ F_0\end{bmatrix}.$$

We can use repeated squaring to compute M^{n-1} by observing that:

- If n is odd, then $M^n = M^{(n-1)/2}\cdot M^{(n-1)/2}\cdot M$
- If n is even, then $M^n = M^{n/2}\cdot M^{n/2}$

Let $T(n)$ be the time to compute M^n; this will be the same as the time to compute M^{n-1} asymptotically. Each matrix multiplication involves 8 scalar multiplications and 4 scalar additions since these are 2×2 matrices. If scalar multiplication and scalar addition take constant time, each matrix multiplication requires constant time. Hence

$$T(n) = T(n/2) + c \Rightarrow T(n) = O(\log n).$$

Once we compute M^{n-1}, computing F_n takes constant extra time (*why?*). Hence the time to compute F_n is $O(\log n)$.

(c) Note that F_n has at least $n/2$ bits but fewer than n bits (easy to prove inductively that $F_n < 2^n$). Hence addition of two Fibonacci numbers smaller than F_n requires $O(n)$ time.

i. The recurrence becomes

$$T(n) = T(n-1) + T(n-2) + cn = \Omega(n2^{n/2}).$$

ii. There are n iterations. Each iteration takes time $O(n)$, yielding $O(n^2)$ total time to compute F_n.

iii. Let $T(n)$ be the time to compute M^n. Each matrix multiplication involves 8 scalar multiplications and 4 scalar additions; multiplying two $O(n)$-digit numbers requires $O(n^2)$ time, adding them requires only $O(n)$ time. Hence

$$T(n) = T(n/2) + cn^2 = O(n^2).$$

It follows that the time to compute F_n is $T(n) + O(n^2) = O(n^2)$.

Algorithm 2

`fib`(n) //n should be non-negative

```
1: if i = 0 then
2:     return 0;
3: end if
4: if i = 1 then
5:     return 1;
6: end if
7: a =fib(i − 2);
8: b =fib(i − 1);
9: result = a + b;
10: return result;
```

Algorithm 3

`iter-fib`(n) //n should be non-negative

```
1: if n ≤ 1 then
2:     return n;
3: end if
4: a = 0;
5: b = 1;
6: for i = 2 to n do
7:     result = a + b;
8:     a = b;
9:     b = result;
10: end for
11: return result;
```

Algorithm 4 Pseudo-code for FIB

```
1: function FIB(n)
2:     if n = 0 then
3:         return 0
4:     end if
5:     if n = 1 then
6:         return 1
7:     end if
8:     M ← [[1, 1], [1, 0]]
9:     result ← MATRIXPOWER(M, n − 1)
10:    return result[0][0]
11: end function
12:
13: function MATRIXPOWER(M, n)
14:    if n = 1 then
15:        return M
16:    end if
17:    result ← POWER(M, n div 2)
18:    if n mod 2 = 0 then
19:        result ← result ∗ result
20:    else
21:        result ← result ∗ result ∗ M
22:    end if
23:    return result
24: end function
```

RECOMMENDED EXERCISES (do NOT return, they will not be graded)

1. **Solution to recommended exercise 1**

   - According to master theorem, $a = 4$, $b = 2$, $k = 3$. Thus $a < b^k$ and $T(n) = O(n^k)$, hence $T(n) = O(n^3)$.
   - According to master theorem, $a = 8$, $b = 2$, $k = 2$. Thus $a > b^k$ and $T(n) = n^{\log_b a}$, hence $T(n) = O(n^3)$.
   - According to master theorem, $a = 6$, $b = 3$, $k = 1$. Thus $a > b^k$ and $T(n) = n^{\log_b a}$, hence $T(n) = O(n^{\log_3 6})$.
   - This recurrence is not in the form required by the master theorem. Therefore, we would have to explicitly construct and analyze the recursion tree for this recurrence. Alternatively, we may use the following substitutions.

     Let $n = 2^m$ (hence $n = \log n$). Then

     $$T(2^m) = T(2^{m/2}) + 1.$$

     This recurrence is still not in the desired form but it's quite close. Let $F(m) = T(2^m)$, thus

     $$F(m) = F(m/2) + 1.$$

     This is the recurrence we encountered for the running time of binary search. Or, we can use the Master Theorem to obtain $F(m) = O(\log m)$. It follows that

     $$T(n) = T(2^m) = F(m) = O(\log m) = O(\log \log n).$$

2. **Solution to recommended exercise 2**

   First note that $f(n) = \sum_{i=0}^{n} \lambda^i$. Then for $n \geq 1$:

   (a) $\lambda < 1$:

   Then $f(n) \geq 1$ and $f(n) < \sum_{i=0}^{\infty} \lambda^i = \frac{1}{1-\lambda}$.

   Hence $f(n) = \Omega(1)$ and $f(n) = O(1)$, thus $f(n) = \Theta(1)$.

   (b) $\lambda = 1$

   Then $f(n) = \sum_{i=0}^{n} \lambda^i = n + 1$. Hence $f(n) = \Theta(n)$.

   (c) $\lambda < 1$

   Then $f(n) = \frac{\lambda^{n+1}-1}{\lambda-1}$, and $f(n) \geq \lambda^n$ and $f(n) \leq \frac{\lambda}{\lambda-1}\lambda^n$. Hence $f(n) = \Theta(\lambda^n)$.

3. Solution to problem 3

f	g	O	o	Ω	ω	Θ
$\log^2 n$	$6 \log n$	n	n	y	y	n
$\sqrt{\log n}$	$(\log \log n)^3$	n	n	y	y	n
$4n \log n$	$n \log 4n$	y	n	y	n	y
$n^{3/5}$	$\sqrt{n} \log n$	n	n	y	y	n
$5\sqrt{n} + \log n$	$2\sqrt{n}$	y	n	y	n	y
$n^5 4^n$	5^n	y	y	n	n	n
$\sqrt{n} 2^n$	$2^{n/2+\log n}$	n	n	y	y	n
$n \log 2n$	$\frac{n^2}{\log n}$	y	y	n	n	n
$n!$	2^n	n	n	y	y	n
$\log n!$	$(\log n)^{\log n}$	y	y	n	n	n

4. *Hint: Think binary search*

5. *Hint: Think D&Q to get an $O(n \log n)$ algorithm. Next, can you come up with a linear-time algorithm for this problem?*