CSOR W4246 - Summer 2021
Homework #1

Joseph High - `jph2185`

May 15, 2021

Problem 1

The pseudocode for the algorithm described in the problem is given below.

```
1:  TwoThirdsSort(A, i, j):
2:      d = length(A[i : j]) + 1
3:      if d == 2 then
4:          if A[i] > A[j] then
5:              Swap A[i] and A[j]
6:          end if
7:      else if d > 2 and i < j then
8:          k = ceiling(2/3 × d)
9:          TwoThirdsSort(A, i, i + k)          // 1st phase: sorting first ⌈2/3⌉ of A
10:         TwoThirdsSort(A, j − k, j)          // 2nd phase: sorting last ⌈2/3⌉ of A
11:         TwoThirdsSort(A, i, i + k)          // 3rd phase: sorting first ⌈2/3⌉ of A
12:     end if
13:     return A
14: end
```

Correctness: Induction on n.

Base case ($n = 1$ and $n = 2$): For $n = 1$, the input list consists of one item; the algorithm returns the list with one element, which is clearly sorted. For $n = 2$, there are two cases: if the element in the first position is less than the element in the second position, the algorithm returns the list as is; otherwise, the algorithm swaps the position of each element, so that the list is now sorted. In either case, the algorithm returns a sorted list.

Induction Hypothesis: For all $0 \leq m \leq n - 1$, assume `TwoThirdsSort` correctly sorts on input size m.

Inductive Step: Consider running `TwoThirdsSort` on input size n. In the first, second, and third phase, the algorithm sorts sub-lists of size $\lceil \frac{2}{3}n \rceil$. By the induction hypothesis, since $\lceil \frac{2}{3}n \rceil \leq n - 1$, the algorithm correctly sorts the sub-lists in each of these phases. Clearly, this implies that the entire list is then sorted.

Running Time: Let $T(n)$ be the running time of the `TwoThirdsSort` algorithm. The running time to compare the elements of lists of size 2 is $O(1)$. Each call to `TwoThirdsSort` on the first $\lceil 2/3 \rceil$ of the elements takes time $T(2n/3)$, by our definition of $T(n)$. Similarly, the running time for `TwoThirdsSort` on the last $\lceil 2/3 \rceil$ of the elements of the list is $T(2n/3)$. The total running time is then:

$$\begin{aligned} T(n) &= T(2n/3) + T(2n/3) + T(2n/3) + c \\ &= 3 \cdot T(2n/3) + c \end{aligned}$$

Then, $a = 3,\ b = \frac{3}{2}$, and $k = 0 \implies a > b^k$. Therefore, by the Master Theorem, the running time is

$$T(n) = O(n^{\log_{3/2} 3}) = O(n^{2.71})$$

Because the running time of this algorithm is significantly worse than that of other sorting algorithms, one would not use this algorithm in their next application to sort. For example, the running time of mergesort, insertion sort, and quicksort have (worst-case) running times of $O(n \log n)$, $O(n^2)$, and $O(n^2)$, respectively. This algorithm has a running time of $O(n^{2.71})$, which is significantly worse than the running time of each of the listed sorting algorithms.

Problem 2

(a) Lines 1, 2, 7, and 8 all run in constant time. The for-loop at lines 3 through 6 iterates n times, and the comparison made within the for-loop has a constant time run time for each iteration. Therefore, the running time of this algorithm is $O(n)$.

(b) The `Randomized Approximate Median` algorithm randomly generates a number according to the discrete uniform distribution on S which is then used to approximate the median of S. Because the output is probabilistically correct (in addition to potentially returning an error), the `Randomized Approximate Median` algorithm is a Monte Carlo algorithm.

Regarding the success probability, a number is uniformly sampled from S. Because the probability of generating any of the n numbers uniform across S, the probability that 1/4 of the numbers in S are less than a_i is approximately 1/4. It's not exactly 1/4 because we have to exclude a_i. Specifically, the probability is $\frac{n-1}{4} \cdot \frac{1}{n} = \frac{1}{4} - \frac{1}{4n}$.

Similarly, the probability that 1/4 of the numbers in S are greater than a_i is $\frac{n-1}{4} \cdot \frac{1}{n} = \frac{1}{4} - \frac{1}{4n}$. Therefore, the probability that the algorithm returns a number (and not `error`) is

$$\frac{1}{4} - \frac{1}{4n} + \frac{1}{4} - \frac{1}{4n} = \frac{1}{2} - \frac{1}{2n} = \frac{1}{2}\left(1 - \frac{1}{n}\right)$$

That is, the success probability is $\frac{1}{2} - \frac{1}{2n}$.

(c) (Based on hints provided during office hours) Clearly, you can improve the success probability by running the algorithm multiple times, generating a potentially different value from the set S. In doing so, it reduces the probability that the algorithm returns an error. In one run of the algorithm, the probability that the algorithm returns `error`

is one minus the success probability: $1 - \left(\frac{1}{2} - \frac{1}{2n}\right) = \frac{1}{2} + \frac{1}{2n}$

Running the algorithm, say, k times reduces to probability of an error to $\left(\frac{1}{2} + \frac{1}{2n}\right)^k$ since a_i is independently sampled in each run of the algorithm $\implies$ each run of the algorithm is independent of all other runs. For a sufficient sample size and a sufficient number of runs, the success probability will reach over 99%. The number of runs of the algorithm to achieve a success probability of over 99% depends on the size of the input. The running time of the new algorithm (running the algorithm k times) is then $O(kn)$

Problem 3

(a) *Proof.* For simplicity, assume that a new item passes by at each time step $t \in \{1, 2, \ldots, k\}$. Let X_t denote the random variable that indicates which item is stored at time t, and let the sequence $\{z_1, z_2, \ldots, z_k\}$ denote the all of the items that have passed by at time t. Induct on the number of items that have appeared at time t.

Base case ($t = 1$): At $t = 1$, only the first item has appeared. The algorithm automatically stores the first time. That is, the item is stored with probability 1. Clearly, the storage of a single item in a sample size of one is uniformly distributed. Therefore, the base case is satisfied.

Induction Hypothesis: For all $t \in \{1, \ldots, k\}$, assume the algorithm stores exactly one item $z_i \in \{z_1, z_2, \ldots, z_k\}$ at a time such that $P\{X_t = z_i\} = 1/k$. That is, $X_t \sim Unif\{1, k\}$. [1]

Inductive Step: At time $t = k + 1$, item z_{k+1} appears. The algorithm replaces the current item with z_{k+1} with probability $\frac{1}{k+1}$.

Now, consider the event where any other item $z_i \neq z_{k+1}$ is stored at time $t = k + 1$. This is equivalent to the event where item z_i is not replaced with z_{k+1} AND z_i being stored at time t. That is,

$$\begin{aligned}
P\{X_{k+1} = z_i, \forall i \in \{1, \ldots, k\}\} &= P\left[\{z_i \text{ not replaced with } z_{k+1}\} \cap \{z_i \text{ stored at time } t = k\}\right] \\
&= P\left[\{X_{k+1} = z_i;\ i \neq k+1\} \cap \{X_k = z_i\}\right] \\
&= P\{X_{k+1} = z_i;\ i \neq k+1\} \cdot P\{X_k = z_i\} \\
&= \left(1 - \frac{1}{k+1}\right) \cdot \frac{1}{k} \\
&= \frac{k}{k+1} \cdot \frac{1}{k} \\
&= \frac{1}{k+1}
\end{aligned}$$

Therefore, the algorithm ensures that all items are stored with equal probability. □

[1] $Unif\{1, k\}$ denotes the discrete uniform distribution on $\{1, \ldots, k\}$

(b) Suppose that the currently stored item is z_i, where $i \in \{1, 2, \ldots, k-1\}$, and that z_i is replaced with the k^{th} item, z_k, with probability 1/2. That is, $P\{X_k = z_k\} = 1/2$. Then, the probability that z_k does not replace z_i is also 1/2. For $i < k$, consider the event where z_i is stored at time $t = k$. In other words, z_i was not replaced by z_k at time $t = k$. In fact, if $k - i > 1$, this indicates that z_i was not replaced by any of the items that appeared after $t = i$ and before $t = k$. It is assumed that the event of storing, or not storing, an item at each time step occurs independently. Then for $i \in \{2, \ldots, k\}$, the probability that an item z_i is stored at time t is then:

$$\begin{aligned}
P\{X_k = z_i\} &= P\left[\{z_i \text{ not replaced by } z_{i+1}, \ldots, z_{k-1}, z_k\} \cap \{z_i \text{ replaced item stored at time } t = i\}\right] \\
&= P\{z_i \text{ not replaced by } z_{i+1}, \ldots, z_{k-1}, z_k\} \cdot P\{z_i \text{ replaced item stored at time } t = i\} \\
&= P\left[\{X_{i+1} = z_i\} \cap \{X_{i+2} = z_i\} \cdots \cap \{X_k = z_i\}\right] \cdot P\{X_i = z_i\} \\
&= \left(1 - \frac{1}{2}\right)^{k-i} \cdot \frac{1}{2} \\
&= \left(\frac{1}{2}\right)^{k-i+1}
\end{aligned}$$

For $i = 1$, the item z_1 does not replace any preceding items since it is the first one in the sequence. Therefore, $P\{X_k = z_1\} = \left(\frac{1}{2}\right)^{k-1}$.

Letting f denote the distribution of the stored items, we then have that

$$f(z_i) = \begin{cases} \left(\frac{1}{2}\right)^{k-i+1} & , \quad \text{for } i \in \{2, \ldots, k\} \\ \left(\frac{1}{2}\right)^{k-1} & , \quad \text{for } i = 1 \end{cases}$$

Problem 4

(a) To prove the inequality, we induct on n.

Proof. **Base case** ($n = 6$):

$$\begin{aligned}
F_6 = F_5 + F_4 = F_4 + F_3 + F_4 &= 2(F_3 + F_2) + F_3 \\
&= 3F_3 + 2F_2 \\
&= 3(F_2 + F_1) + 2(F_1 + F_0) \\
&= 3(2F_1 + F_0) + 2F_1 + 2F_0 \\
&= 8F_1 + 5F_0 \\
&= 8 \qquad \text{(since } F_1 = 1 \text{ and } F_0 = 0\text{)} \\
&= 2^3 \;=\; 2^{6/2}
\end{aligned}$$

$\implies \quad F_6 \geq 2^{6/2}$

Therefore, the inequality holds for the base case ($n = 6$).

Induction Hypothesis: Assume $F_k \geq 2^{k/2}$ holds for all $k \in \{6, \ldots, n\}$.

Inductive Step: Consider the $(n+1)^{st}$ Fibonacci number:

$$\begin{aligned}
F_{n+1} &= F_n + F_{n-1} \\
&\geq 2^{n/2} + 2^{\frac{n-1}{2}} && \text{(by the induction hypothesis)} \\
&= 2^{n/2} + 2^{n/2} \cdot 2^{-\frac{1}{2}} \\
&= \frac{2^{n/2} \cdot 2^{\frac{1}{2}} + 2^{n/2}}{2^{\frac{1}{2}}} \\
&= 2^{n/2} \cdot 2^{\frac{1}{2}} \cdot \left(\frac{1 + 2^{-\frac{1}{2}}}{2^{\frac{1}{2}}} \right) \\
&= 2^{\frac{n+1}{2}} \cdot \underbrace{\left(\frac{2^{\frac{1}{2}} + 1}{2} \right)}_{>1} \\
&\geq 2^{\frac{n+1}{2}}
\end{aligned}$$

$$\implies F_{n+1} \geq 2^{\frac{n+1}{2}}$$

□

(b) i.) The pseudocode for the algorithm below computes the n^{th} Fibonacci number, as defined by the given recursion.

```
1:  Fib(n):
2:      Initialize F_n = 0
3:      if n == 0 then
4:          F_n = F_n
5:      else if n == 1 then
6:          F_n = 1
7:      else
8:          F_n = Fib(n − 1) + Fib(n − 2)
9:      end if
10:     return F_n
11: end
```

Running time: Let $T(n)$ denote the running time of the algorithm on input n. Lines 2 through 6 require constant time. Lines 7 through 9 recursively calls `Fib`$(n-1)$

and `Fib`$(n-2)$. That is, it calls the algorithm on input sizes reduced by 1 and by 2, respectively, until reaching either 0 or 1. By our definition of $T(n)$, this gives a run time of $T(n-1)+T(n-2)$. In each recursion, there are addition and subtraction operations, each of which requires $O(1)$ time, by the supposition. Putting it all together, the total (worst-case) running time can be expressed as follows:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

which resembles the Fibonacci recurrence $F_n = F_{n-1} + F_{n-2}$. To determine a lower bound, consider the following:

$$\begin{aligned} T(n) = T(n-1) + T(n-2) + O(1) &\geq T(n-2) + T(n-2) + O(1) \\ &= 2 \cdot T(n-2) + O(1) \\ &\geq 2 \cdot 2 \cdot T(n-4) + O(1) \\ &= 2^2 \cdot T(n-4) + O(1) \\ &\geq 2^3 \cdot T(n-6) + O(1) \\ &\vdots \\ &\geq 2^{\frac{n-1}{2}} \cdot T(n-(n-1)) = 2^{\frac{n-1}{2}} \cdot T(1) \\ &\geq 2^{n/2} \end{aligned}$$

Hence, $T(n) \geq 2^{n/2}$. Therefore, a lower bound for the running time is $T(n) = \Omega(2^{n/2})$.

Correctness: Induction on n.

Base case ($n = 0$ and $n = 1$): For $n = 0$, the condition in the first if-statement in the `Fib` procedure is a true statement, so the `Fib`(0) returns 0, which is consistent with the Fibonacci number for $n = 0$. For $n = 1$, the condition in the second if-statement is true, while all others are false, returning the value of 1 for `Fib`(1), which is consistent with the Fibonacci number for $n = 1$.

Induction Hypothesis: Assume the algorithm computes the correct Fibonacci number for all $k \in \{2, \ldots, n\}$.

Inductive Step: Consider running the algorithm on input $n+1$. For $n \geq 2$, `Fib`$(n+1)$ computes `Fib`(n) + `Fib`$(n-1)$. By the induction hypothesis, both `Fib`(n) and `Fib`$(n-1)$ return the correct n^{th} and $(n-1)^{st}$ Fibonacci numbers, respectively. Because the sum of the n^{th} and $(n-1)^{st}$ Fibonacci numbers is, by definition, the $(n+1)^{st}$ Fibonacci number, the algorithm correctly computes the $(n+1)^{st}$ Fibonacci number.

ii.) The algorithm uses a quasi-memoization technique by storing previously computed values in a temporary variables. In particular, the algorithm utilizes two temporary variables to store the Fibonacci numbers computed in the previous two iterations to compute the Fibonacci number in the next iteration. In doing so, it reduces the number of addition operations by eliminating the need to recompute the Fibonacci numbers of lower degree.

```
1:  Fib2(n):
2:      Initialize F_{k-1} = 0
3:      Initialize F_k = 1
4:      if n == 0 then
5:          return F_{k-1}
6:      else if n == 1 then
7:          return F_k
8:      else
9:          for k = 2 to n do
10:             F_{k+1} = F_k + F_{k-1}        // computes the k^{th} Fibonacci number
11:             F_k = F_{k+1}                  // temporarily stored to use in next iteration
12:             F_{k-1} = F_k                  // temporarily stored to use in next iteration
13:         end for
14:     end if
15:     return F_{k+1}
16: end
```

Running Time: Lines 2 through 8 are all computed in constant time. The for-loop at lines 9 through 13 iterates $n-1$ times. Therefore, the total running time is $O(n)$.

iii.) We can use a divide and conquer approach to compute F_n in the desired running time. Using the hint, consider the following:

$$\begin{aligned}
\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \cdot \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \cdot \begin{bmatrix} F_{n-3} \\ F_{n-4} \end{bmatrix} \\
&\vdots \qquad\qquad \vdots \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}
\end{aligned}$$

Since $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ is a square matrix, we can divide the problem of computing $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ into smaller sub-problems as follows:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{\frac{n-1}{2}} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{\frac{n-1}{2}}$$

which ultimately only consists of multiplication and addition operations, which

require constant time to compute.

Running time: Letting $T(n)$ be the (worst-case) running time of the algorithm. Because we are recursively dividing the problem of computing the matrix product by 2, the running time is $T(n) = T(n/2)$. Then, $a = 1,\ b = 2,\ k = 0 \implies a = b^k$. By the Master Theorem, $T(n) = O(\log n)$

Correctness: Induct on n.
Base case: For $n = 2$, we have

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_1 + F_0 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1+0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

which is consistent with the 1^{st} and 2^{nd} Fibonacci numbers. Hence, the base case is satisfied.

Induction Hypothesis: For all $1 \leq k \leq n$, assume the algorithm computes F_k correctly.

Inductive Step: Consider the following to compute F_{n+1}:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n + F_{n-1} \\ F_n \end{bmatrix}$$

Because this is a recursive algorithm, F_n and F_{n-1} are computed before computing F_{n+1}. From the above, $F_{n+1} = F_n + F_{n-1}$ which is consistent with the Fibonacci recurrence. By the induction hypothesis, the algorithm computes F_n and F_{n-1} correctly. Therefore, the algorithm correctly computes the $(n+1)^{st}$ Fibonacci number.