CSE548, AMS542: Analysis of Algorithms, Fall 2019 **Date:** October 16, 2019

Midterm Exam

(7:00 PM – 8:15 PM : 75 Minutes)

- This exam will account for either 15% or 30% of your overall grade depending on your relative performance in the midterm and the final. The higher of the two scores (midterm and final) will be worth 30% of your grade, and the lower one 15%.
- There are three (3) questions, worth 75 points in total. Please answer all of them in the spaces provided.
- There are 16 pages including five (5) blank pages and two (2) pages of appendices. Please use the blank pages if you need additional space for your answers.
- The exam is *open slides* and *open notes*. But *no books* and *no computers*.

Good Luck!

Question	Pages	Score	Maximum
1. Moving Average	2–3		25
2. SuperStoogeSort	5–6		25
3. Fast Fib	8–11		25
Total			75

Name: _____

Question 1. [25 Points] Moving Average. A *moving average* can reveal longer-term trends in time series data by smoothing out short-term fluctuations. They are widely used in economics and financial applications as well as in science and engineering.

This question asks you to efficiently compute a moving average of a given time series data.

1(a) **[7 Points]** Suppose you are given a sequence of n values $v_1, v_2, \ldots, v_n$, where v_t is the data value at time step $t \in [1, n]$. We assume that $v_t = v_0$ for all time steps $t < 1$.

An m-point *simple moving average* (SMA) at time step $t \in [1, n]$, where $m \in [1, n]$, is:

$$SMA_m(t) = \frac{v_{t-(m-1)} + v_{t-(m-2)} + \ldots + v_{t-1} + v_t}{m} = \frac{\sum_{i=t-(m-1)}^{t} v_i}{m}.$$

Given $m \in [1, n]$, show that $SMA_m(t)$ for all time steps $t \in [1, n]$ can be computed in $\Theta(n)$ time.

1(b) [**18 Points**] Consider the time series data from part 1(a).

An m-point *weighted moving average* (WMA) at time step $t \in [1, n]$ is:

$$WMA_m(t) = \frac{v_{t-(m-1)}w_{m-1} + v_{t-(m-2)}w_{m-2} + \ldots + v_{t-1}w_1 + v_t w_0}{w_{m-1} + w_{m-2} + \ldots + w_1 + w_0} = \frac{\sum_{i=t-(m-1)}^{t} w_{t-i}v_i}{\sum_{i=0}^{m-1} w_i},$$

where $m \in [1, n]$, and w_i is the weight given to the data point located $i \in [0, m-1]$ times steps to the left of time step t.

Give an algorithm to compute $WMA_m(t)$ for all time steps $t \in [1, n]$ in $\Theta(n \log n)$ time.

Use this page if you need additional space for your answers.

Figure 1: "The Three Stooges" Curly, Larry, and Moe produced more than 90 hugely popular short comedy films between 1934 and 1946.

```
SuperStoogeSort( A[1 : n] )  {Sort the numbers in array A[1 : n]
                               in nondecreasing order of value.}
 1. if n < 30 then
 2.     sort A[1 : n] using any standard sorting algorithm
 3. else
 4.     SuperStoogeSort( A[1 : ⌈4n/9⌉] )          {Larry sorts}
 5.     SuperStoogeSort( A[1 : ⌈2n/3⌉] )          {Moe sorts}

 6.     SuperStoogeSort( A[n − ⌈8n/27⌉ + 1 : n] )  {Curly sorts}
 7.     SuperStoogeSort( A[n − ⌈4n/9⌉ + 1 : n] )   {Larry sorts}
 8.     SuperStoogeSort( A[n − ⌈2n/3⌉ + 1 : n] )   {Moe sorts}

 9.     SuperStoogeSort( A[1 : ⌈4n/9⌉] )          {Larry sorts}
10.     SuperStoogeSort( A[1 : ⌈2n/3⌉] )          {Moe sorts}
```

Figure 2: Sort the numbers in $A[1 : n]$ in nondecreasing order of value.

Question 2. [25 Points] SuperStoogeSort. The SuperStoogeSort algorithm shown in Figure 2 sorts an array $A[1 : n]$ of n numbers in nondecreasing order of value. The algorithm is correct though more inefficient than the already inefficient StoogeSort[1] algorithm from which it is derived. Each of the steps 4–10 is associated with either Moe or Larry or Curly from the comedy "The Three Stooges" (see Figure 1). It turns out that the algorithm will still be correct if the steps associated with Larry and Curly are dropped, but one cannot say that for any of the steps associated with Moe.

Now answer the following questions.

2(*a*) [**7 Points**] Write the recurrence relation describing the running time $T(n)$ of SuperStoogeSort($A[1 : n]$). Ignore all ceilings for simplicity (that is, replace $\lceil x \rceil$ with x for every $x \in \left\{ \frac{2n}{3}, \frac{4n}{9}, \frac{8n}{27} \right\}$).

[1] Problem 7-3, "Introduction to Algorithms," 2nd edition, by Cormen, Leiserson, Rivest, and Stein

2(*b*) [**18 Points**] Solve your recurrence from part 2(*a*) to show that $T(n) = \Theta\left(n^{\log_{1.5}\left(\frac{1}{\sqrt[3]{2}-1}\right)}\right)$. Use the Akra-Bazzi method[2].

[2] Turns out that $T(n) = \omega\left(n^{3.3}\right)$ and $T(n) = o\left(n^{3.33}\right)$.

Use this page if you need additional space for your answers.

Question 3. [**25 Points**] **Fast Fib.** Fibonacci numbers are defined by the following recurrence, where f_n is the n^{th} Fibonacci number:

$$f_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ f_{n-1} + f_{n-2} & \text{if } n > 1. \end{cases}$$

This question asks you to analyze the performance of two algorithms for computing f_n for large n.

3(a) [**5 Points**] Show that the binary representation of f_n has $\Theta(n)$ bits.

3(*b*) [**5 Points**] Argue that the SlowFib algorithm shown in Figure 3 takes $\Theta\left(n^2\right)$ time to compute the n^{th} Fibonacci number.

```
SlowFib( n )    {Return the n^th Fibonacci number.}
1.  if n < 2 then
2.      return max{0, n}
3.  else
4.      f'' ← 0,  f' ← 1                {f'' = f_0 and f' = f_1}
5.      for i ← 2 to n do
6.          f ← f' + f''                {f_i = f_{i-1} + f_{i-2}}
7.          f'' ← f'
8.          f' ← f
9.      return f                        {f = f_n}
```

Figure 3: Compute the n^{th} Fibonacci number.

3(c) [**5 Points**] The FastFib algorithm in Figure 4 is based on the observation that $f_1 = f_1$ and $f_2 = f_0 + f_1$ can be written in matrix form as follows: $\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$.

Then $\begin{bmatrix} f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$, and $\begin{bmatrix} f_3 \\ f_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^3 \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$, etc.

In general, $\begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$.

```
FastFib( n )    {Return the n^th Fibonacci number.}
1. if n < 2 then
2.     return max{0, n}
3. else
4.     f_0 ← 0,  f_1 ← 1
5.     [ f_n ; f_{n+1} ] ← [ 0 1 ; 1 1 ]^n [ f_0 ; f_1 ]
6.     return f_n
```

Figure 4: Compute the n^{th} Fibonacci number.

Show that FastFib(n) can compute f_n for $n > 1$ using no more than $14 \log_2 n$ integer multiplications.

3(d) [**10 Points**] Prove that the FastFib algorithm shown in Figure 4 can compute f_n in $\Theta\left(n^{\log_2 3}\right)$ time.

Use this page if you need additional space for your answers.

Use this page if you need additional space for your answers.

Use this page if you need additional space for your answers.

Appendix: Recurrences

Master Theorem. Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1, \\ aT\left(\frac{n}{b}\right) + f(n), & \text{otherwise,} \end{cases}$$

where, $\frac{n}{b}$ is interpreted to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ has the following bounds:

Case 1: If $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.

Case 2: If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some constant $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.

Case 3: If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Akra-Bazzi Recurrences. Consider the following recurrence:

$$T(x) = \begin{cases} \Theta(1), & \text{if } 1 \leq x \leq x_0, \\ \sum_{i=1}^{k} a_i T(b_i x) + g(x), & \text{otherwise,} \end{cases}$$

where,

1. $k \geq 1$ is an integer constant,
2. $a_i > 0$ is a constant for $1 \leq i \leq k$,
3. $b_i \in (0, 1)$ is a constant for $1 \leq i \leq k$,
4. $x \geq 1$ is a real number,
5. x_0 is a constant and $\geq \max\left\{\frac{1}{b_i}, \frac{1}{1-b_i}\right\}$ for $1 \leq i \leq k$, and
6. $g(x)$ is a nonnegative function that satisfies a polynomial growth condition (e.g., $g(x) = x^\alpha \log^\beta x$ satisfies the polynomial growth condition for any constants $\alpha, \beta \in \Re$).

Let p be the unique real number for which $\sum_{i=1}^{k} a_i b_i^p = 1$. Then

$$T(x) = \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right).$$

Appendix: Computing Products

Integer Multiplication. Karatsuba's algorithm can multiply two n-bit integers in $\Theta\left(n^{\log_2 3}\right) = \mathcal{O}\left(n^{1.6}\right)$ time (improving over the standard $\Theta\left(n^2\right)$ time algorithm).

Matrix Multiplication. Strassen's algorithm can multiply two 2×2 matrices using 7 multiplications, and two $n \times n$ matrices in $\Theta\left(n^{\log_2 7}\right) = \mathcal{O}\left(n^{2.81}\right)$ time (improving over the standard $\Theta\left(n^3\right)$ time algorithm).

Polynomial Multiplication. One can multiply two n-degree polynomials in $\Theta\left(n \log n\right)$ time using the FFT (Fast Fourier Transform) algorithm (improving over the standard $\Theta\left(n^2\right)$ time algorithm).

Appendix: Closed Form for the n^{th} Fibonacci Number (f_n)

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Appendix: Computing Powers using Repeated Squaring

```
POW( X, n )                                          {Return X^n for integer n > 0.}
1.  if n = 1 then
2.      return X
3.  else
4.      m ← ⌊n/2⌋
5.      Y ← POW( X, m )                              {recursively compute Y = X^m}
6.      Y ← Y · Y          {multiply Y = X^m and Y = X^m to get Y = X^{2m}}
7.      if 2m ≠ n then                               {if n is odd (i.e., n = 2m + 1)}
8.          Y ← X · Y      {multiply X and Y = X^{2m} to get Y = X^{2m+1}}
9.      return Y                                     {return Y = X^n}
```

Figure 5: Compute X^n by repeated squaring.