A Multiple Choice Questions ($13 \times 3 = 39$ marks)

Select the **best unique** answer for each question. Each correct answer worth 3 marks.

1. In Lecture 1, you learned about the recursive implementation to compute the n-th Fibonacci number modulo m: `RFIB(n, m)`. You have been shown that the number of instructions executed by `RFIB(n, m)` is $\geq 2^{(n-2)/2}$. With this information and other analysis of `RFIB(n, m)`, you conclude that the time complexity of `RFIB(n, m)` is **not**:

   a). $\Omega(2^{(n-2)/2})$

   b). $\Omega(2^{n/2})$

   ~~c). $\Omega(2^n)$~~

   d). $O(2^n)$

   e). $O(3^n)$

2. Let $f(n) = 50n + 7\log n + 1$, we want to prove that $f(n) = O(n \log n)$. Now, recall the definition of $O(\cdot)$ notation. What should be the chosen c and n_0?

   a). $c = 1$, $n_0 = 16$

   b). $c = 7$, $n_0 = 1$

   c). $c = 7$, $n_0 = 49$

   d). $c = 7$, $n_0 = 1024$

   e). $c = 50$, $n_0 = 2$

   $50 \cdot 1024 + 7 \cdot 10 + 1$

   $7 \cdot 1024 \cdot 10$

   $50 \cdot 2$ $\quad$ $50 \cdot 2 \cdot 1$

   $7 \cdot 1$

   1

   Linked List

3. Suppose that you run the first two phases of Counting Sort pseudo-code on a *fictional* computation model where accessing a location i in any array takes $c \cdot i$ operations for constant $c \geq 1$ (but all other operations are exactly like the normal Word-RAM model). For simplicity, array A contains single-digit integers, i.e., $k = 10$, $0 \leq A[i] < k$, $\forall i \in [0..n-1]$.

```
for i <- 1 to k
    do C[i] <- 0
for j <- 1 to n
    do C[A[j]] <- C[A[j]]+1
```

   1 to 10

   What is the tightest $O(\cdot)$ time complexity of the code above?

   a). $O(n^3)$

   b). $O(n^2)$

   c). $O(n \log n)$

   d). $O(n)$

   e). $O(\log n)$

4. Let $f(n) = 5n + 1024 + 77n^2 + \sin(n) + \cos(n + \pi/2)$.
Which statement is **incorrect**?

a). $f(n) = o(n^2)$

b). $f(n) = O(n^2 \log^2 n)$

c). $f(n) = \Theta(n^2)$

d). $f(n) = \Omega(n)$

e). $f(n) = \omega(n \log n)$

5. Rank the following recurrences in **increasing** order of growth; that is, put $T_x(n)$ before function $T_y(n)$ in your list, only if $T_x(n) = O(T_y(n))$ (there is no ties).

- $T_1(n) = 2 \cdot T_1(n/4) + 1$ $\log_b a = 1/2$ $\Theta \sqrt{n}$
- $T_2(n) = T_2(n/1.1111111) + 1$ $\lg n$
- $T_3(n) = 2 \cdot T_3(n/4) + \sqrt{n} \log^2 n$ $\sqrt{n} \lg^3 n$
- $T_4(n) = 8 \cdot T_4(n/2) + n^3 \cdot \sqrt{n}$ $n^{3.5}$

a). T_1, T_3, T_4, T_2

b). T_1, T_2, T_3, T_4 $T_2 T_1 T_3 T_4$

c). T_2, T_1, T_3, T_4

d). T_2, T_1, T_4, T_3

e). T_2, T_3, T_1, T_4

6. You are given the following pseudo-code:

```
// assume that there are arrays: A and MA, both with n integers (0-based index)
MA[0] <- A[0]
for i <- 1 to n-1
    do MA[i] <- max(MA[i-1], A[i])
```

We want to use this pseudo-code to compute the maximum integer for each prefix of array A. What should be the loop invariant to prove the correctness of this pseudo-code?

a). `MA[i]` contains the maximum integer in `A[0..i]`, $\forall i \in [0..n-1]$

b). `i` is always less than n

c). `MA[i]` values are sorted in non-decreasing order, $\forall i \in [0..n-1]$

d). `MA[i]` values are greater than or equal to `A[0]`, $\forall i \in [0..n-1]$

e). `MA[i]` values are positive integers

7. There is an impostor among the five algorithm names below. Select the impostor.
Hint: Four algorithms are randomized algorithms.

   a). Strassen's (matrix multiplication) algorithm
   b). Freivalds' (matrix multiplication verification) algorithm
   c). QuickSelect algorithm
   d). Welzl's (smallest enclosing circle) algorithm
   e). Karger's (min-cut) algorithm

8. We are using a comparison-based sorting algorithm to sort 7 distinct integers in non-decreasing order. The minimum number of comparisons needed to guarantee correct sorted output:

   a). 12
   b). 13
   c). 14
   d). 15
   e). 16

$7! = 5040$

$\lceil {}^2\log 5040 \rceil = 13$

9. You want to sort n 64-bit *non-negative* signed integers in non-decreasing order ($1 \leq n \leq 7777$). Which algorithm uses the least number of operations in the Word-RAM model for this scenario?

See lec?

   a). Merge sort ??
   ~~b). Counting sort~~
   c). Radix sort with $r = 32$ bits ($64/32 = 2$ passes of stable counting sort)
   d). Radix sort with $r = 21$ bits ($63/21 = 3$ passes of stable counting sort)
   e). Radix sort with $r = 16$ bits ($64/16 = 4$ passes of stable counting sort)

$\lg 7777 = 13 \rightarrow 100517$

$2(n + 2^{32})$

65536

$7777 + 64 \cdot 1024$

$4(n + 2^{16})$

10. Recall, while analyzing the randomized Quick sort in the lecture, we use the notation e_i to denote the i-th smallest element in the input array. Let Y_{ij} be the indicator random variable where $Y_{ij} = 1$ if element e_i is compared with element e_j during Randomized Quick Sort of array A, or $Y_{ij} = 0$ otherwise. What is $\mathbb{E}[Y_{ij}]$ for $i = 10$ and $j = 19$?

   a). 0.1
   b). 0.2
   c). 0.4
   d). 77%
   e). 100%

$\frac{2}{j-i+1} = \frac{2}{10}$

11. You have not studied the first half of CS3230 topics. Thus, for each of the 5 options of the MCQs in this paper, you randomly select any option. You do this random choice independently for each question. There is no negative marking for wrong answer and 3 marks for correct answer. What is the expected score for this 13 MCQs (Section A) if you use that strategy?

a). 2.6

b). 7.7 $13 \times 3/5$

c). 7.8

d). 9.75

e). 13

12. You want to use Freivalds' algorithm (that creates an $n \times 1$ random 0/1 vector) to check if $A \cdot B = C$ (where each of A, B, C are large square matrices). You run Freivalds' algorithm 77 times because you are afraid of making mistake. However, for all those runs, Freivalds' algorithm keeps saying that $A \cdot B = C$. Which statement is the **most logical**?

a). It is very unlikely that Freivalds' algorithm makes a mistake after that many iterations

b). For the given A, B, and C, $A \cdot B = C$ with 100% certainty

c). For the given A, B, and C, $A \cdot B \neq C$ with 100% certainty

d). The expected runtime of Freivalds' algorithm varies for each of those 77 iterations

e). The error rate of Freivalds' algorithm decreases linearly per each iteration

13. You were impressed with the *worst-case* linear time selection algorithm presented in the lecture of Week 06. You implemented and used it for this semester's Programming Assignment 2 (PA2), task B. However, you kept getting the dreaded Time Limit Exceeded verdict. Which statement is the **most logical**?

~~a).~~ Dr Diptarka's analysis during the lecture on Week 06 was wrong, that algorithm perhaps takes more than linear time

~~b).~~ Dr Steven's time limit setup is too strict as the worst-case $O(n)$ selection algorithm is the current best possible algorithm for this semester's PA2-B task

c). We need to change the groups of 5 presented in Lecture 6 to the groups of 3

d). We need to change the groups of 5 presented in Lecture 6 to the groups of 7

e). The *worst-case* linear time selection algorithm likely gets the Time Limit Exceeded verdict due to its large hidden constant factor in its $O(n)$ asymptotic analysis

National University of Singapore

School of Computing

CS3230 - Design and Analysis of Algorithms

Midterm Test

(Semester 2 AY2022/23)

Time Allowed: 90 minutes

INSTRUCTIONS TO CANDIDATES:

1. Do **NOT** open this assessment paper until you are told to do so.
2. This assessment paper contains TWO (2) sections.
   It comprises ELEVEN (11) printed pages, including this page.
3. This is an **Open Book** Assessment.
4. For Section A, use the OCR form provided (use 2B pencil).
   You will still need to hand over the entire paper as the MCQ section will not be archived.
5. For Section B, answer **ALL** questions within the **boxed space**.
   If you leave the boxed space blank, you will get automatic 1 mark (even for Bonus question).
   However, if you write at least a single character and it is totally wrong, you will get 0 mark.
   You can use either pen or pencil. Just make sure that you write **legibly**!
6. Important tips: Pace yourself! Do **not** spend too much time on one (hard) question.
   Read all the questions first! Some questions might be easier than they appear.
7. You can assume that all **logarithms are in base** 2.
8. Please write your Tutorial Group, '_', and Student Number only. Do **not** write your name.

T	1	3	_	A	0	2	3	6	4	3	9	X

This portion is for examiner's use only

Section	Maximum Marks	Your Marks	Grading Remarks
A	39	36	
B	61	64	=) (including bonus)
Total	100	100	=)

B Essay Questions (61 marks)

B.1 Prove that case 3 regularity condition is always satisfied in PA1-A (14 marks)

In Programming Assignment 1 (PA1), task A, we have to use master theorem to automatically solve recurrences in the form of:

$$T(n) = a \cdot T(\tfrac{n}{b}) + c \cdot n^d \log^k n$$

We are also given the following constraints $a > 0$, $b > 1$, $c > 0$, $d \geq 0$, and $k \geq 0$.

When case 3 of master theorem is applicable, many students do *not* also check if the required regularity condition is also satisfied, yet all of them still get the Accepted verdict (assuming there is no other bug other than skipping regularity condition check on case 3 situations). This is not because the test cases are weak. In fact, the regularity condition is *always* satisfied for case 3 of master theorem in this semester's PA1. Your job in this question is to formally prove it.

14

We assume Case 3 of master theorem holds for this particular f: then we have $f = \Omega(n^{\log_b(a)+\varepsilon})$ for some ε. Consequently we have

$c \cdot n^d \lg^k n \geq C_0 \cdot n^{\lg_b(a)+\varepsilon}$ for all $n \geq n_0$ ($\exists C_0 \in \mathbb{R}^+$).

This implies $n^{d-\lg_b(a)-\varepsilon} \lg^k n \geq \frac{C_0}{c}$ $\forall n \geq n_0$.

Now, we must have $d > \lg_b(a)$. Otherwise, if $d - \lg_b(a) \leq 0$, then for any $\varepsilon > 0$, we must have $\lim_{n\to\infty} n^{\underbrace{d-\lg_b(a)}_{\leq 0}-\varepsilon} \lg^k n$

$\leq \lim_{n\to\infty} 1 (= n^0)\, n^{-\varepsilon} \log^k n = 0$ since

if $n = 2^{n'}$, we have $\frac{n \cdot n'}{2^{n' \cdot \varepsilon}} \to 0$ as $n' \to +\infty$ (linear vs exponential growth). So, there is no possible C_0 to pick as $\lim \to 0$. Conclusively, we have

$d > \lg_b(a)$; i.e. $b^d > a$ or $\frac{a}{b^d} < 1$.

Now we prove regularity condition ($\exists c < 1$ s.t. $\frac{a f(\frac{n}{b})}{f(n)} < c$ $\forall n > N$, $\exists N$.)

LHS is $\frac{a \cdot (\frac{n}{b})^d \log^k(\frac{n}{b})}{n^d \log^k n} = \frac{a}{b^d} \left(\frac{\log n - \log b}{\log n}\right)^k$. Since $\lim_{n\to\infty} \frac{\lg n - \lg b}{\lg n} = 1$, so $\lim_{n\to\infty} \left(\frac{\lg n - \lg b}{\log n}\right)^k = 1$ and we can pick

$$1 > C = \frac{\left(\frac{a}{b^d}+1\right)}{2} > \frac{a}{b^d}$$

such that $\left(\frac{\log n - \log b}{\log n}\right)^k < \frac{C}{(a/b^d)} < 1$ for all $n > N$ for large.

ok

B.2 Exponentiation in Addition Machine (35 marks)

We have learned that we count the number of instructions the algorithm takes to measure its running time. If you recall our first lecture, we consider the Word-RAM as our computation model because this model resembles our modern computers. However, what about old computers with a more primitive computation model?

In 1990, Robert Floyd and Donald Knuth investigated a computation model called Addition Machine. This model only has the following limited arithmetic instructions:

- Addition (+)
- Subtraction (−)
- Comparisons $(=, \neq, <, \leq, >, \geq)$

Simply put, this model is equivalent to any modern language (e.g., C++, Java, or Python) but **WITHOUT** using multiplication, division, modulo, exponentiation, or even bit-wise operations. You can assume that this Addition Machine model is a restricted Word-RAM model.

So if we want to multiply $x \times y$ or divide (and round down) $\lfloor x/y \rfloor$, we can naively implement them as follows:

- For multiplication, we can repeatedly increment a temporary variable by x, for y many times, assuming $y \leq x$. If $x < y$, then we swap the x and y first, thus this $MULTI(x, y)$ runs in $\Theta(min(x, y))$ time.
- For division (with round down), we can repeatedly decrement x by y as long as x stays non-negative. The number of repetitions will be the answer, thus this $DIV(x, y)$ takes $\Theta(x/y)$ time.

B.2.1 Naive Exponentiation (5 Marks)

Suppose that we want to implement an exponentiation function a^n naively by multiplying a for n many times as the following:

$a \times (a \times (a \times \ldots (a \times a)) \ldots)$

```
int NAIVE_EXP(int a, int n) {
    if (n == 0) return 1;
    else return MULTI(a, NAIVE_EXP(a, n-1));
}
```

What is the time complexity, in $\Theta(\cdot)$ notation, of the above function?
For simplicity, assume that $a \leq n$ and the inputs are non-negative.

Since $\min(a, a^k) = a \ \forall k \geq 1$, we have NaiveExp$(n)$ = a + NaiveExp$(n-1)$.
By telescoping or recursion tree we get NaiveExp$(n, a) = \Theta(na)$
$= O(n^2)$ since $a \leq n$.

a is hidden above since it is invariant.

Note: if $a = \Theta(n)$, then $\Theta(na) = \Theta(n^2) = \Theta(a^2)$.

5/5

B.2.2 Fast Exponentiation? (14 Marks)

You have learned from our past lecture that there is a "faster" algorithm for exponentiation using a Divide and Conquer technique like the following:

```
int FAST_EXP(int a, int n) {
    if (n == 0) return 1;
    else if (n == 1) return a;
    else {                                      O(n)
        int temp = FAST_EXP(a, DIV(n, 2));
        temp = MULTI(temp, temp)
        if (IS_ODD(n)) temp = MULTI(a, temp);
                  O(n)                      O(a)
        return temp;
                  O(1)
    }
}
```

Even by assuming that the *IS_ODD* function takes $O(n)$ time, you might think that this algorithm should run faster than the naive one, **but is it really true**? Prove (or disprove) by finding the time complexity, in $\Theta(\cdot)$ notation, of the *FAST_EXP* function! For simplicity, assume that $a \leq n$, n is a power of 2, and the inputs are non-negative.

14/14

There is a very very humongous bottleneck at temp = MULTI (temp, temp)
This is because runtime of MULTI(x, x) where x is int $== \Theta(x)$.
However, at the last iteration of FAST-EXP when temp = FASTEXP(a, n/2),
we have temp $== a^{\lfloor n/2 \rfloor}$ as an integer, so runtime is at least
$\Omega(a^{\lfloor n/2 \rfloor})$, even worse than naive implementation.

Now, complete analysis is as follows:

$F\_E(n) = FE(n/2) + \Theta(a^{n/2}) + \Theta(n+a)$ $\quad (\Theta(n) \in O(a^{n/2}))$ if $a > 2$, n large enough

fast Exp

$= \cancel{FE(n/4) + \Theta(a^{n/4}) + \Theta(a^{n/2})}$

$= \cancel{\ldots = FE(1) + \Theta(a^1) + \Theta(a^2) + \ldots} + \Theta(a^{n/4}) + \Theta(a^{n/2})$.

For Θ, need to argue both ≤ and ≥.

We prove that $FE(n) = \Theta(a^{n/2})$. Let we have $FE(k) \leq a^{k/2}c$ for $k < n$.

Now, $FE(n) \leq c \cdot a^{n/4} + 1 \cdot a^{n/2}$
$= c\sqrt{a^{n/2}} + a^{n/2}$
$\leq c \cdot a^{n/2}$. So, induction is complete

We choose c to handle all small cases

We have $cx + x^2 < cx^2$ for $cx < (c-1)(x^2)$ or $x > \frac{c}{c-1} > 2$, and we must have $a^{n/2} > 2$ for large n
$\quad\; x$

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

$T(2k+1) =$

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

$n^2 = (n-1)^2 + 2n - 1$

 $\Theta(1) + \lg n$.

B.2.3 Squaring a Number (14 Marks)

While it is not trivial, actually there is a faster division implementation such that $DIV(x, y)$ runs in only $\Theta(\log(x/y))$ time. We can then use it to implement $IS\_ODD(n)$ also in $\Theta(\log n)$ time. You don't need to prove them.

Instead, your job is to implement the algorithm of $SQUARE(n)$. This algorithm should return the value of n^2, and you may assume that n is always non-negative. You must also analyze the time complexity of your algorithm (using $\Theta(\cdot)$ notation). You may call the naive $MULTI$, the faster DIV, and the faster IS_ODD functions in your implementation.

To get a full mark, your algorithm should run in $O(\log^2 n)$ time, and you need to provide a correct $\Theta(\cdot)$ analysis. Partial 3 marks will be given for any algorithm which runs in $\Theta(n)$ time. **HINT**: Use Divide and Conquer technique!

```
Algo: ~~perform~~ int k=IS_ODD(n)   # k==1 if n odd (?)
  if (~~IS_ODD(n)~~ k):          Base case? -2
     int half = DIV(n-1,2)
     int four_half_square = MULTI(4, SQUARE(half))
     int four_half = MULTI(4, half)
     return four_half_square + four_half + 1     ✓ +5
  else:
     int half = DIV(n,2)
     return MULTI(4, SQUARE(half))     ✓ +5
```

Analysis: IS_ODD $= \lg n$, DIV $= \lg(n/2) = \Theta(\lg n)$. +4

four half square is the bottleneck; it runs in $T(n-1/2) + 4$ time.

for simplicity assume $n-1/2 == n/2$ in C++ (so $n/2 - n/2$ if n even)

actual math ... C++ ... actual math C++

square would've run in $\Theta(1)$ if we use a "modern" implementation.

$T(n) = T(n/2) + \underbrace{\Theta(\lg n) + \Theta(1)}_{\Theta(\lg n)}$

By master's case 2 we have $T(n) = \lg^{1+1}(n) = \lg^2 n$ ✓

12

B.2.4 Lesson Learned (2 Marks)

Lastly, please write anything that you learned from these small "experiments"! :)

Basic operations need to be $\Theta(1)$ for all numbers for any algorithm to follow "standard/expected runtime/behavior". I.e. don't try to "break abstraction" and always use libraries (where people have implemented what you intended better & faster ~~than what you will probably do). It's better to rely on smarter people than to reinvent the wheel yourself~~ to do. ✓ 2

(or can just use recursion tree:

$T(n) = T(n/2) + \log n = T(n/4) + \log n/2 + \log n$

$= \ldots = \lg 1 + \ldots + [(\log n) - 1] + \lg n$

$= \Theta\left(\frac{\log^2 n}{2}\right) = \Theta(\log^2 n)$

(B.2.3. continued)

Worst case is $n = 2^k - 1 = 2^{k-1} + 2^{k-2} + \ldots + 1$, and that is also $\Theta(k) + \Theta(k-1) + \ldots + \Theta(1) = \Theta(k^2) = \Theta(\log^2 n)$.

B.3 Moderately Small Element (12 marks)

Given an unsorted array of n integers, we know how to find the smallest element in $O(n)$ time, and also we have learned in the lecture that $\Omega(n)$ time is necessary for this purpose. Now suppose we aim to find a *moderately small* element (instead of the smallest element). For an n-length array A, we call the element $A[i]$ *moderately small* if its rank is at most $n/10$. (Recall, if $A[i]$ is the j-th smallest element in the array A, then its rank is j.)

Given an unsorted array of length n, our objective is to find a moderately small element in $o(n)$ time with the help of randomization. For that purpose, try to solve the following questions.

B.3.1 What is the Probability (I)? (2 marks)

Let us pick an index i uniformly at random from the set $\{0, 1, \cdots, n-1\}$, and return $A[i]$. What is the probability that the returned output is a moderately small element?

$1/10$. To be exact, there are $\left\lfloor \frac{n}{10} \right\rfloor$ moderately elements ~~of~~ the set, and so probability is $\frac{\lfloor n/10 \rfloor}{n} \approx \frac{1}{10}$.

2

B.3.2 What is the Probability (II)? (7 marks)

Let us now modify the procedure described in subsection B.3.1 as follows: Pick a *sequence* of indices $i_1, i_2, \cdots, i_s$ uniformly at random and independently from the set $\{0, 1, \cdots, n-1\}$. Then output the *smallest* element among the set $\{A[i_1], A[i_2], \cdots, A[i_s]\}$. What is the probability that the returned output is a moderately small element?

Assuming probability of choosing moderately small element is $p \left(:= \frac{1}{10}\right.$, if answer in B.3.1. is $1/10$ instead of $\lfloor n/10 \rfloor / n \left.\right)$,

the probability that $\min(\{A[i_1], \dots, A[i_s]\})$ has rank $\leq n/10$ is

$1 - \Pr[\min \text{ has rank} > n/10]$

~~$= 1 - \Pr[\text{every ele of } \{A[i_1], \dots, A[i_s]\} \text{ has rank} > n/10]$~~

$= 1 - (1-p)^s = 1 - \left(\frac{9}{10}\right)^s \quad \left(\text{if } p = \frac{1}{10}\right).$

7

B.3.3 What is the Running Time? (3 marks)

Given any n-length array, if you want to output a moderately small element with probability at least $1 - \frac{1}{n}$, what would be the tightest $O(\cdot)$ bound on the time complexity of the procedure described in subsection B.3.2?

$1 - \left(\frac{9}{10}\right)^S > 1 - \frac{1}{n} \rightarrow \left(\frac{9}{10}\right)^S < \frac{1}{n}$ or $\left(\frac{10}{9}\right)^S > n$.

We then must have $S > {}^{10}\!/_{9} \log n = {}^{10}\!/_{9} \log e \cdot \ln n = \Theta(\ln n)$

$(= O(\lg n)$, too$)$.

3

B.4 Bonus Question (5 marks)

Suppose you are given as input a circular array $A[0 \cdots n-1]$ of length n containing all the distinct integers between $\{1, 2, \cdots, n\}$ in an arbitrary order. Your goal is to decide whether there exists three consecutive indices $i, i+1, i+2$ such that $A[i] + A[i+1] + A[i+2] > 1.5 \cdot n$. (Note, since the input array is circular, you should actually consider $i + 1(\mod n)$ and $i + 2(\mod n)$.) So if there exists such three consecutive indices, you should output "YES"; otherwise "NO". How many cells of the input array A you must read (in the worst-case) to output the correct answer? (Provide proper explanation in support to your answer.) [**No partial marks** will be given for this question.]

Zero. We prove that regardless of order, always exist such i. So always output "YES".
Let $S(i) = A[i] + A[i+1 \bmod n] + A[i+2 \bmod n]$. We know $\sum_{i=0}^{n-1} S(i) = 3\sum_{j=1}^{n} j$ since each number $1 \leq j \leq n$ appears in 3 $S(i)$s. So, $\sum S(i) = \frac{3n(n+1)}{2}$.
We prove by contradiction. Let all $S(i)$ satisfy $S(i) \leq 1.5n = \frac{3}{2}n$. So, $\sum S(i) \leq \frac{3}{2}n^2$, a contradiction from the fact that $\sum S(i) = \frac{3}{2}n^2 + \frac{3}{2}n$.

5

– END OF PAPER; All the Best –

~~$\frac{3n(n-1)}{2} / n = \frac{3(n-1)}{2} = 1.5(n-1)$~~
~~$= 1.5n - 1.5$~~

~~All is $1.5n - 1$~~
or $1.5n$ (?)