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			_	A	0							

This portion is for examiner's use only

Section	Maximum Marks	Your Marks	Grading Remarks
A	39		
B	61		
Total	100		

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

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

The default answers (as the MCQ section is not supposed to be 'publicly' archived to open up possibilities of reuse in the future): cdbac aabab cae.

The rest of this page 1 is REDACTED.

This page 2 is REDACTED.

This page 3 is REDACTED.

This page 4 is REDACTED.

This page 5 is REDACTED.

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.

The regularity condition says that we need to find a constant $C < 1$ such that $a \cdot f(n/b) \leq C \cdot f(n)$.
Warning: This C is different from the c in $c \cdot n^d \log^k n$ in PA1-A chosen $f(n)$ format.

Our $f(n)$ in PA1 is specifically in this format: $c \cdot n^d \log^k n$, so:
$a \cdot c \cdot (\frac{n}{b})^d \log^k \frac{n}{b} \leq C \cdot c \cdot n^d \log^k n$ (expand $f(n)$)
$a \cdot (\frac{n}{b})^d \log^k \frac{n}{b} \leq C \cdot n^d \log^k n$ (remove c from both sides)
$\frac{a}{b^d} \cdot n^d (\log^k n - \log^k b) \leq C \cdot n^d \log^k n$ (rearrange formulas)
$\frac{a}{b^d} \cdot n^d \log^k n \leq C \cdot n^d \log^k n$ (simplify inequality by removing $log^k b$ (positive, as $b > 1$ and $k \geq 0$))
$\frac{a}{b^d} \leq C$ (remove $n^d \log^k n$ from both sides)
So, in PA1-A chosen $f(n)$ format, choosing $C = \frac{a}{b^d}$ is a good choice.
But is this C always < 1?
(you cannot jump to the conclusion because the question says so; but many students did)...
e.g., what if $a = 100$, $b = 2$, and $d = 1$? If we set $C = \frac{100}{2^1} = \frac{100}{2} = 50$, then $C > 1$...
Up to here 7 marks...

To complete the proof, consider when case 3 of master theorem is applicable:
We have $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$.
Or $c \cdot n^d \log^k n = \Omega(n^{\log_b(a)+\epsilon})$.
Or $d > \log_b(a)$ (n^d grows faster than $\log^k n$).
Raising b to the power of LHS and RHS, we have $b^d > b^{\log_b(a)}$, or $b^d > a$, or $1 > \frac{a}{b^d}$, or $\frac{a}{b^d} < 1$.

Now, combining these two information, we have $C \geq \frac{a}{b^d}$ (we can just set $C = \frac{a}{b^d}$) and this $\frac{a}{b^d} < 1$.
This way, the regularity condition is always satisfied for this specific $f(n)$ in PA1-A.

So, we can use this proof to skip the regularity check if our Divide and Conquer algorithm's $f(n)$ is of $c \cdot n^d \log^k n$ type. And good news: This form of $f(n)$ appears (very) frequently in many real Divide and Conquer algorithms...

This task (B.1) is set by Dr Steven Halim, after realizing that his regularity condition assertion checks are always true (hence the "not applicable" case due to failing regularity condition on case 3 cannot be triggered in the chosen $f(n)$ and constraints).

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.

We hide these two pseudocode to save a bit of reading time during exam, but if you are interested, here are the possible implementations:

```
int MULTI(int a, int b) {
    if (a > b) return MULTI(b, a);
    else if (a == 0) return 0;
    else return b + MULTI(a-1, b);
}

int DIV(int a, int b) {
    if (a < b) return 0;
    else return 1 + DIV(a-b, b);
}
```

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:

```
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.

$T(0) = \Theta(1)$
$T(1) = \Theta(min(a, 1)) = \Theta(1)$, then $T(1) = \Theta(1)$.

We now analyze when $n > 1$.
$T(n) = T(n-1) + \Theta(min(a, X))$ (where $X = a^{n-1}$).
So, $T(n) = T(n-1) + \Theta(a)$ (because $X \geq a$ for $n > 1$).

We can expand the recursion tree. Its height is $n - 1$ and at each level, we do exactly $\Theta(a)$ operations (except at the last level, where we do $\Theta(1)$ operations).
Overall, we do: $\Theta(an)$ operations.

PS 1: Since $a \leq n$, we get $\Theta(n^2)$ for the worst-case a.
PS 2: We need Θ analysis, not just Big O or just Ω.

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

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.

$T(n) = T(n/2) + f(n)$ // $f(n)$ depends whether $IS\_ODD(n)$ is true or not.
(up to here is 1 mark, same as leaving this blank)

Since we assume that n is a power of 2, we can ignore the case when $IS\_ODD(n)$ is true.
So, let's just focus on the case when $IS\_ODD(n)$ is false (n is even).

For either case actually, we do $MULTI(temp, temp)$ that is **not** $O(1)$ in *Word-RAM* model, but much slower in the *Addition Machine* model. Here $temp = a^{\frac{n}{2}}$.
So, the multiplication done by $MULTI(temp, temp)$ requires $\Theta(min(a^{\frac{n}{2}}, a^{\frac{n}{2}})) = \Theta(a^{\frac{n}{2}})$.
(observing that $MULTI(temp, temp)$ is the dominating operation in terms of runtime and deriving the recursion correctly gives 7 marks)

OPTIONAL: If $IS\_ODD(n)$ is true, we need one more multiplication, i.e., $\Theta(min(a, a^{n-1})) = \Theta(a)$.
This extra multiplication in the odd case is negligible compared to the multiplication of two big *temp* values in the even case, so $f(n) = \Theta(a^{\frac{n}{2}} + a) = \Theta(a^{\frac{n}{2}})$.
(the analysis of odd cases is not part of the marking scheme; up to here still 7 marks)

We now have $T(n) = T(n/2) + \Theta(a^{n/2})$.

This is case 3 of master theorem where the root does the most work, because $n^{log_b a} = 1$ and $f(n) = \Omega(n^{0+\epsilon})$ for $\epsilon > 0$.
But the $f(n)$ is not like Section B.1 earlier, so we have to do a proper regularity condition check
(penalty of 4 marks if students skipped this regularity check on custom $f(n) = c'a^{n/2}$):
$1 \times c'a^{(n/2)/2} \leq c \times c'a^{n/2}$
$a^{n/4} \leq c \times a^{n/2}$
$\frac{a^{n/4}}{a^{n/2}} \leq c < 1$, the regularity condition is satisfied.
So, we have $T(n) = \Theta(a^{n/2})$.

Thus, we see that $FAST\_EXP$ runs asymptotically slower (now exponential!) than $NAIVE\_EXP$ in the *Addition Machine* model.
(using master theorem/other valid analysis is +7 marks; up to here is full 14 marks).

P.S. 1: Simplifying this to $T(n) = \Theta(a^n)$ is wrong.
P.S. 2: Since $a \leq n$, we get $\Theta(n^{n/2})$ for the worst-case a.
P.S. 3: The above can also be solved using other methods such as recursion tree, but note that the sum obtained *does not result in a geometric series.*
(-4 marks for incorrect use of other methods, resulting in a different answer)

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!

For an easy 3 partial marks (better than leaving this blank), one can just use the naive $MULTI(n, n)$.
Its time complexity is $\Theta(min(n, n)) = \Theta(n)$...

```
int SQUARE_NAIVE(int n) {
    return MULTI(n, n);
}
```

Also, using $FAST\_EXP$ from subsection B.2.2 does not help.
Its time complexity is $\Theta(n^{\frac{2}{2}}) = \Theta(n)$...

```
int SQUARE_NAIVE(int n) {
    return FAST_EXP(n, 2);
}
```

So it is better to do this instead:

```
int SQUARE(int n) {
    if (n == 0) return 0;
    else {
        int temp = MULTI(4, SQUARE(DIV(n, 2)));
        if (IS_ODD(n)) temp = temp + MULTI(2, n) - 1;
        return temp;
    }
}
```

Explanation: Again, we encounter whether $IS\_ODD(n)$ is true (n is even) or not (n is odd).
But this time, we need to handle both cases.
Either way, we will have to do $SQUARE(n) = MULTI(4, SQUARE(DIV(n, 2)))$.
This way, we divide n by 2 recursively until $n = 0$ (base case), which we return trivially as $0^2 = 0$.
Admittedly, not many of us will naturally square a number in this Divide and Conquer way...
This part has complexity of $\Theta(min(4, X))$ where X is a larger integer + $\Theta(\log(n/2))$.
Overall, this part is $\Theta(\log n)$.
(This creative D&C recurrence worth 5 marks, so up to here 5 marks).

Now if $IS\_ODD(n)$ (n is odd), we need to do an adjustment:
$n^2 = ((n-1)+1)^2 =$
$(n-1)^2 + 2 \times (n-1) \times 1 + 1^2 =$
$(n-1)^2 + 2 \times n - 2 + 1 =$
$(n-1)^2 + 2 \times n - 1,$
And if n is odd, it means $n-1$ is even and we can use the previous result.
So if $IS\_ODD(n)$, we add the previous result by adjustment of $MULTI(2,n) - 1$.
Again, this adjustment is negligible as $\Theta(min(2,n)) = \Theta(2) = \Theta(1)$.
(This adjustment worth 5 marks, so up to here 10 marks).

Now the analysis: We have $T(n) = T(n/2) + \Theta(\log n + 1)$, we combine both cases.
So $T(n) = T(n/2) + \Theta(\log n)$.
This is case 2 of master theorem: $\log_2 1 = 0$ and the power of n is also 0.
$c \cdot n^0 \log n = \Theta(n^0 \log^1 n)$, notice that $k = 1$, so $T(n) = \Theta(log^2 n)$.
(using master theorem/other valid analysis is +4 marks; up to here is full 14 marks)

PS 1: Be careful not to forget handling the base case ($N = 0$), else -2 marks.
PS 2: We need Θ analysis too here, not just O (so need to be careful with recursion tree here).
PS 3: There is an alt-solution which simultaneously solves $MULTI(x,y)$ in $\Theta(\log^2(min(x,y)))$:)

B.2.4 Lesson Learned (2 Marks)

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

Open ended, e.g., we usually take it for granted that all basic computer operations are $O(1)$ while it is actually not the case. In fact, even multiplication is not precisely $O(1)$ in our current modern Word-RAM model. We should keep in mind that we can still use the concepts taught in CS3230 to design and analyze the complexity of any algorithm in any model of computation. :)

This task (B.2) is set by TA Ammar Fathin Sabili, based on his PhD thesis.

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?

There are $n/10$ ranks out of n ranks that is considered as moderately small elements.
We will get that moderately small element randomly with probability $\frac{n/10}{n} = \frac{1}{10}$. (You will get a slightly different term if you consider floor/ceiling.)

Grading scheme: No partial mark.

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 $\{0, 1, \cdots, n-1\}$ with replacement. 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?

Step 1: The probability that we *do not* get moderately small element per try is $1 - \frac{1}{10} = \frac{9}{10}$.
Step 2: The probability that we *do not* get moderately small element after s tries is $(\frac{9}{10})^s$.
Step 3: The probability that we get at least one moderately small element after s tries is $1 - (\frac{9}{10})^s$.

If someone solves it by assuming the samples were drawn "without replacement", he/she will also get the full marks.

Grading scheme: Upto step 1: +2, upto step 2: +5, upto step 3: +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?

We want $1 - (\frac{9}{10})^s \geq 1 - \frac{1}{n}$, how much repeat ($s$) that we need to do in terms of n?

$$1 - (\frac{9}{10})^s \geq 1 - \frac{1}{n} \Leftrightarrow (\frac{10}{9})^s \geq n \Leftrightarrow s \geq \log_{10/9} n.$$

So it suffices to set $s = \log_{10/9} n$, which leads to the running time of $O(\log n)$. (A similar bound holds even if someone assumes "without replacement" in subsection B.3.2.)

Grading scheme: Upto first inequality: +1, for writing s such that $(10/9)^s \geq n$: +2.

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.]

No cells need to be read. The reason is as follows:

Observe that $\sum_{i=1}^{n} A[i] = \sum_{k=1}^{n} k = n(n+1)/2$ (since A contains all the distinct integers between $\{1, 2, \cdots, n\}$). Now consider any starting index i and take the consecutive indices $i, i+1 (\mod n), i+2 (\mod n)$, and let $S_i := A[i] + A[i+1] + A[i+2]$. Then

$$\sum_{i=1}^{n} S_i = 3 \sum_{i=1}^{n} A[i] = \frac{3n(n+1)}{2}.$$

Then by averaging, there must exist an index i such that

$$S_i \geq \frac{3n(n+1)}{2n} > 1.5n.$$

Hence, the output should always be "YES".

Grading scheme: No partial mark.

The last two tasks (B.3 and B.4) are by Dr Diptarka Chakraborty.

– END OF PAPER; All the Best –