COMP 3711 – Design and Analysis of Algorithms
2021 Fall Semester – Written Assignment # 1
Distributed: Sept 9'th 2021
Due: Sept 20'th 2021, 11:59 PM
Solutions

Your solution should contain
(i) your name, (ii) your student ID #, and (iii) your email address
at the top of its first page.

Some Notes:

- Please write clearly and briefly. Your solutions should follow the guidelines given at
  https://canvas.ust.hk/courses/38226/pages/assignment-submission-guidelines

  In particular, your solutions should be written or printed on *clean* white paper with no watermarks, i.e., student society paper is not allowed.

- Please also follow the guidelines on doing your own work and avoiding plagiarism as described on the class home page.
  You must acknowledge individuals who assisted you, or sources where you found solutions. Failure to do so will be considered plagiarism.

- The term *Documented Pseudocode* in questions (4) and (5) means that your pseudocode must contain documentation, i.e., comments, inside the pseudocode, briefly explaining what each part does.

- Many questions ask you to explain things, e.g., what an algorithm is doing, why it is correct, etc. To receive full points, the explanation must also be *understandable* as well as correct.

- Please make a *copy* of your assignment before submitting it. If we can't find your submission, we will ask you to resubmit the copy.

- Submit a SOFTCOPY of your assignment to Canvas by the deadline. The softcopy should be one PDF file (no word or jpegs permitted, nor multiple files).

  If your submission is a scan of a handwritten solution, make sure that it is of high enough resolution to be easily read. At least 300dpi and possibly denser.

Problem 1: [25 pts]

In class, we "solved" the Mergesort recurrence

$$\forall n > 1, \quad T(n) \le T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \quad \text{and} \quad T(1) = 1 \tag{1}$$

by simplifying and assuming that n was a non-negative integral power of two. This permitted replacing Equation (1) with

$$T(n) \le 2T(n/2) + n \quad \text{and} \quad T(1) = 1.$$

We were then able to use the expansion method (with implicit induction) to derive the *exact* solution

$$T(2^k) \le 2^k + k2^k. \tag{2}$$

If $n = 2^k$ this becomes $T(n) \le n + n \log_2 n$.

We then jumped to say that Equation (2) implies $T(n) = O(n \log n)$.

It doesn't, formally. The expansion method only really solved it for n being a power of 2.

The solution to this homework problem will show, at least for two recurrences, why such simplifications are "legal" and why Equation (1) implies $T(n) = O(n \log n)$ for all $n \ge 1$.

Before attempting this homework, you should first read the "Practice Homework" on the class tutorial page and understand the solution given for that.

That will not only provide some starting intuition for how to solve this problem, it will also illustrate how to formally write solutions.

(a) Let $c > 0$ be constant. Let $T(n)$ be a function satisfying

$$\forall n > 2, \quad T(n) \le T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c \quad \text{and} \quad T(1) = T(2) = 1. \qquad (3)$$

(i) Prove using the expansion method that $T(3^k) = O(k)$.

(ii) Let $S(n)$ be some **nondecreasing** function of n.
You are told that $S(n)$ also satisfies $S(3^k) = O(k)$.
Prove that this implies $S(n) = O(\log_3 n)$.

(iii) For all $n \ge 1$, set $R(n) = \max_{1 \le i \le n} T(i)$. Prove that

$$\forall n > 2, \quad R(n) \le R\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c.$$

(iv) Prove that if $T(n)$ satisfies Equation (3) then $T(n) = O(\log n)$.
You can either do this by using the facts proven in parts (i), (ii) and (iii) (recommended) or directly (much longer).

(b) We learned in class that if $T(n)$ satisfies Equation (1) then it satisfies Equation (2) so

$$T(2^k) = O(k2^k). \qquad (4)$$

(v) Let $S(n)$ be some **nondecreasing** function of n that also satisfies Equation (4). Prove that this implies $S(n) = O(n \log_2 n)$.

(vi) For all $n \ge 1$, set $R(n) = \max_{1 \le i \le n} T(i)$.
Prove that if $T(n)$ satisfies Equation (1) then

$$\forall n > 1, \quad R(n) \le R\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + R\left(\left\lceil \frac{n}{2} \right\rceil\right) + n.$$

(vii) Assuming the correctness of Equation (4), prove that $T(n)$ defined by Equation (1) satisfies $T(n) = O(n \log_2 n)$.
You can either do this by using the facts proven in parts (v) and (vi) (recommended) or directly (much longer).

Requirements, Hints and Comments:

- Prove each of parts (i) - (vii) separately in order.
  Make sure that you clearly label each part.

  Leave a few empty lines between each part.
  Solutions not following this format will have points deducted.
- You should assume the correctness of earlier parts when solving later ones. For example, when solving part (iv), you should assume the correctness of parts (i), (ii) and (iii).
- k and n will always denote non-negative integers.
- $F(n)$ being a *non-decreasing function* means $\forall n,\ F(n) \leq F(n+1)$.
- $f(n) = O(g(n))$ means that

  there exists $c > 0$ *and* n_0 *such that* $\forall n > n_0,\ f(n) \leq cg(n)$.

  Your proofs in parts (i), (ii) (iv), (v) and (vii) should explicitly identify the values of c and n_0 used.
- $T(3^k) = O(k)$ and $T(2^k) = O(k2^k)$ mean, respectively, that

  there exists $c > 0$ *and* k_0 *such that* $\forall k > k_0,\ T(3^k) \leq ck$.

  there exists $c' > 0$ *and* k_0' *such that* $\forall k > k_0',\ T(2^k) \leq c'k2^k$.
- In part (ii) you only know that $S(n)$ is a non-decreasing function satisfying $S(3^k) = O(k)$. Your proof may not use any other assumptions about $S(n)$.

  Hint: Use the fact that for $x \geq 3$, $\log_3 3x = 1 + \log_3 x \leq 2\log_3 x$.
- Similarly, in part (v), you may only use the facts that $S(n)$ is a non-decreasing function satisfying $S(2^k) = O(k2^k)$.
- Write full, formal solutions.

Solution:

[(i)] Let $n = 3^k$ for $k \geq 0$. Then

$$T(3^k) \leq T\left(\left\lfloor \frac{3^k}{3} \right\rfloor\right) + c = T\left(3^{k-1}\right) + c.$$

Applying the expansion method

$$\begin{array}{rcl} T(3^k) & \leq & T(3^{k-1}) + c \\ & \leq & \big(T(3^{k-2}) + c\big) + c = T(3^{k-2}) + 2c \\ & \leq & \ldots \\ & \leq & T(3^{k-i}) + ic \\ & \leq & \ldots \\ & \leq & T(3^{k-k}) + kc \\ & = & 1 + kc. \end{array}$$

This implies $T(3^k) = O(k)$.

More formally, $T(3^k) = O(k)$ because we have shown that

$$\forall k > k_0, \quad T(3^k) \leq 1 + kc \leq 2kc$$

where $k_0 = \lceil \frac{1}{c} \rceil$.

[**(ii)**] Let c', k_0 be such that such $\forall k > k_0$, $S(3^k) \le c'k$.

Note: The fact that $S(3^k) = O(k)$, implies the existence of such c' and k_0. We use c' instead of c so as not to confuse it with the c from Equation (3). Without loss of generality, we may assume that $k_0 > 0$.

- Let $n \ge 3^{k_0}$ and set $k_n = \lceil \log_3 n \rceil$ (implying $k_n \ge k_0$).
- Then $3^{k_n - 1} < n \le 3^{k_n}$ or $k_n - 1 \le \log_3 n \le k_n$.
- **Because $S(n)$ is non-decreasing,**

$$\begin{aligned}\forall n \ge 3^{k_0}, \quad S(n) &\le S(3^{k_n}) && \text{(because } n \text{ is nondecreasing)}\\ &\le c'k_n && \text{(because } S(3^k) = O(k))\\ &\le c'(\log_3 n + 1) && \text{(from definition of } k_n)\\ &\le 2c' \log_3 n. && \text{(because } n \ge 3_{k_0} \ge 3)\end{aligned}$$

- Then $S(n) = O(\log_3 n)$ because we have shown that

$$\forall n > n_0, \quad S(n) \le c'' \log_3 n$$

where $c'' = 2c'$ and $n_0 = 3^{k_0}$.

Marking Notes:
Many students did not understand what was assumed and what needed to be proven. The ONLY things that could be assumed were that
(a) for all n, $S(n) \le S(n+1)$ and that
(b) $S(3^k) = O(k)$ or, equivalently, there exists c', k_0 be such that such $\forall k > k_0$, $S(3^k) \le c'k$.

Marking Note (1iia):
The solution explicitly needed to show where it was using the non-decreasing property (it's impossible to prove this without using that property).

As an example, consider

$$S(n) = \begin{cases} \log_3 n & \text{if } \log_3 n = \lfloor \log_3 n \rfloor \\ n & \text{if } \log_3 n \ne \lfloor \log_3 n \rfloor \end{cases}$$

Note that this $S(n)$ is an example of a function that satisfies $S(3^k) = O(k)$ and does not satisfy $S(n) = O(\log n)$.

This does not contradict the problem because $S(n)$ is NOT non-decreasing.

Marking Note (1iib)
To get full marks for this problem it was necessary to explicitly state that $S(3^k) = O(k)$ implies the existence of a c' and then use that c' to show the existence of c'' implying $S(n) = O(\log n)$. Solutions missing these steps were marked at least partially wrong. Note that this proof structure was given to you in the practice homework.

Marking Note (1iic)
To get full marks it was necessary to *explicitly* state values for c'' (and n_0).

Here is an example of a common incorrect solution.

Let $n = 3^k$. We know that $S(n) = O(k)$. Thus

$$\begin{aligned} S(n) &\le ck && (*) \\ &= c\log_3 3^k \\ &= c\log_3 n, \end{aligned}$$

so $S(n) \le c\log_3 n$ which means $S(n) = O(\log_3 n)$.

This "solution '' is totally incorrect. It doesn't use the non-decreasing property and it doesn't actually do anything. If you read it carefully, you'll notice that it's going in circles. Line (*) is only known for $n = 3^k$ where k is a non-negative integer. That's all.

So it has not proven anything for all of the other values of $n \ne 3^k$ which is what the problem was asking about.

[**(iii)**] For $n > 2$,

$$\begin{aligned} R(n) &= \max_{1 \le i \le n} T(i) & (5) \\ &\le \max\left(T(1),\, T(2),\, \max_{3 \le i \le n}\left(T\left(\left\lfloor \frac{i}{3} \right\rfloor\right) + c\right)\right) & (6) \\ &= c + \max_{1 \le i \le \lfloor n/3 \rfloor} T(i) & (7) \\ &= R\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c. & (8) \end{aligned}$$

Line 6 comes from applications of Equation (3) .

Marking Note: (1iiia)

Any solution to this problem HAD to use Equation (3) as well as the definition of $R(n)$. There was no other way to prove this.

[**(iv)**] Let $T(n)$ be as defined in Equation (3) and $R(n)$ as defined in part (iii).

- From part (iii), $R(n)$ satisfies Equation (3) .
  $\Rightarrow$ from part (i), $R(3^k) = O(k)$.

- $R(n) \le \max(R(n), T(n+1)) = R(n+1)$
  $\Rightarrow$ $R(n)$ is a nondecreasing function of n.
  $\Rightarrow$ from part (ii), $R(n) = O(\log n)$.

- Now note that, by definition $T(n) \le R(n)$.
  $\Rightarrow$ Since $R(n) = O(\log n)$, $T(n) = O(\log n)$ as well.

Marking Note: The answer here needed to be EXPLICIT as to how it was using (i), (ii) and (iii).

A common mistake made was assuming that $T(n)$ is non-decreasing. That was not legal. All you knew about $T(n)$ is that it satisfies Equation (3). That absolutely did NOT imply that $T(n)$ is non-decreasing.

As a counterexample, consider the function $T(n)$ that gives the number of non-zero digits in the ternary representation of n. Here are the first 11 values

n	1	2	3	4	5	6	7	8	9	10	11	12	13
$TERN(n)$	1	2	10	11	12	20	21	22	100	101	102	110	111
$T(n)$	1	1	1	2	2	1	2	2	1	2	2	2	3

Note that $T(1) = T(2) = 1$ and, for $n > 1$,

$$T(3n) = T(n), \quad T(3n+1) = T(n)+1, \quad \textbf{and} \quad T(3n+2) = T(n)+2$$

Thus,

$$\forall n > 2, \quad T(n) \le T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + 1 \quad \textbf{and} \quad T(1) = T(2) = 1.$$

This $T(n)$ satisfies Equation (3) but is NOT non-decreasing.

Alternate Proof of (iv):

The intent of this question was for you to use (i), (ii) and (iii) to prove (iv) (and see that it's not necessary for $T(n)$ to be nondecreasing).

It was also possible to skip these steps and prove (iv) directly as follows.

First prove the following lemma.

Lemma: If $n \leq 3^k$ then $T(n) \leq (c+1)k$.

Proof: The proof will be by induction on k.

Note that $T(3) \leq T(1) + c = 1 + c$. So, the statement is true for $k = 1$.

Now assume that the Lemma is true for $k-1$. Note that if $n \leq 3^k$ then $\lfloor n/3 \rfloor \leq 3^{k-1}$. Thus

$$\begin{aligned} T(n) &\leq T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c \\ &\leq (c+1)(k-1) + c \\ &\leq (c+1)k. \end{aligned}$$

■

Since $n \leq 3^{\lceil \log_3 n \rceil}$, the lemma implies that, for $n \geq 3$,

$$T(n) \leq (c+1)\lceil \log_3 n \rceil \leq (c+1)(1 + \log_3 n) \leq (c+1) 2 \log_3 n$$

so $T(n) = O(\log n)$.

[**(v)**] Since $S(2^k) = O(k2^k)$ there exist c', k_0 such that such

$$\forall k > k_0, \quad S(2^k) \le c' k 2^k.$$

- Let $n \ge 2^{k_0}$ and set $k_n = \lceil \log_2 n \rceil$ (implying $k_n \ge k_0$).
- Then $2^{k_n - 1} \le n \le 2^{k_n}$ or, equivalently, $k_n - 1 \le \log_2 n \le k_n$ and $2^{k_n} \le 2n$.
- Because $S(n)$ is non-decreasing (similar to the derivation in (ii))

$$\forall n \ge 2^{k_0}, \quad S(n) \le S(2^{k_n}) \le c' k_n 2^{k_n} \le c'(\log_2 n + 1)2n \le 4c' n \log_2 n.$$

- Thus $S(n) = O(n \log_2 n)$.

Marking Note: See the marking notes for (ii).

[**(vi)**] For $n > 1$,

$$\begin{aligned}
R(n) &= \max_{1 \le i \le n} T(i) && (9)\\
&\le \max\left(T(1), \max_{2 \le i \le n}\left(T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i\right)\right) && (10)\\
&\le n + \max_{2 \le i \le n}\left(T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right)\right) && (11)\\
&\le n + \max_{1 \le i \le \lfloor n/2 \rfloor} T(i) + \max_{1 \le i \le \lceil n/2 \rceil} T(i) && (12)\\
&= n + R\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + R\left(\left\lceil \frac{n}{2} \right\rceil\right). && (13)
\end{aligned}$$

Note that Equation (10) comes from multiple applications of Equation (1)

Equation (11) comes from using $i \le n$.

Equation (12) comes from the fact that, for all functions $f(m)$ and $g(m)$

$$\max_{\ell \le i \le r}(f(i) + g(i)) \le \max_{\ell \le i \le r} f(i) + \max_{\ell \le i \le r} g(i).$$

Marking Note: See the marking notes for (iii).

[**(vii)**]

Let $T(n)$ be as defined in Equation (1) and $R(n)$ as defined in part (vi).

- From (vi), $R(n)$ satisfies Equation (1) .
  $\Rightarrow$ from the fact given in class, $R(2^k) = O(k2^k)$.
- By definition, $R(n)$ is a nondecreasing function of n.
  $\Rightarrow$ from Part (v) $R(n) = O(n \log n)$.
- Now note that, by definition $T(n) \le R(n)$.
  $\Rightarrow$ Since $R(n) = O(n \log n)$, $T(n) = O(n \log n)$ as well.

Marking Note: See the marking notes for (iv).

<u>Alternate Proof:</u>

The intent of this question was for you to use (iv), (v) to prove (vi) (and see that it's not necessary for $T(n)$ to be nondecreasing).

It was also possible to prove (vi) directly as follows.

First prove the following lemma.

Lemma: If $n \le 2^k$ then $T(n) \le 2k2^k$.

Proof: The proof will be by induction on k.

Note that $T(2) \le 2T(1) + 2 = 4$. So, the statement is true for $k = 1$.

Now assume that the Lemma is true for $k-1$. Note that if $n \le 2^k$ then $\lfloor n/2 \rfloor \le \lceil n/2 \rceil \le 2^{k-1}$. Thus, for $n \le 2^k$,

$$\begin{aligned} T(n) &\le T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \\ &\le 2(2(k-1)2^{k-1}) + n \\ &\le 2(2(k-1)2^{k-1}) + 2^k \\ &= 2(k-1)2^k + 2^k \\ &= 2k2^k - 2^k < 2k2^k. \end{aligned}$$

■

Note that $\lceil \log_2 n \rceil \le 1 + \log_2 n$ and $2^{\lceil \log_2 n \rceil} < 2n$.

Since $n \le 2^{\lceil \log_2 n \rceil}$, the lemma implies that, for $n \ge 2$,

$$\begin{aligned} T(n) &\le 2\lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil} \\ &\le 2(1 + \log_2 n)2n \\ &\le 2(2\log_2 n)2n = 8n\log_2 n \end{aligned}$$

so $T(n) = O(n \log n)$.

Problem 2 [14 pts]

For each pair of expressions (A, B) below, indicate whether A is O, Ω, or Θ of B. Note that zero, one, or more of these relations may hold for a given pair. List the most appropriate relation.

It often happens that some students will get the directions wrong, so please write out the relation in full, i.e., you should write exactly one of the terms or write that *none of the relations is satisfied*.

More explicitly, if any of the relationships are satisfied, you should write the appropriate

$$A = O(B), \quad A = \Omega(B), \quad \text{or} \quad A = \Theta(B)$$

and not just $O(B)$, $\Omega(B)$ or $\Theta(B)$, omitting the A.

If no relationship is satisfied write *none of the relations is satisfied.*

(a) $A = n^3 - n^2 \log n,\ B = 30n^2 - 7n^3 + 2n^4;$

(b) $A = \log_{100}((n+2)!),\ B = \log_2 n^n;$

(c) $A = (16)^{\frac{1}{\log_n 5}},\ B = 2^{\sqrt{3 \log_2 n}};$

(d) $A = \sum_{k=1}^{n} k^4,\ B = 10 \times \binom{n}{5};$

(e) $A = n^6 2^{n(\log_2 n)^3},\ B = n^8 + 2021^{2020^{2019}};$

(f) $A = \sum_{k=1}^{n} \frac{1}{k(k+1)},\ B = \ln\left(\sum_{k=1}^{n} \frac{1}{k}\right);$

(g) $A = n(1 + (-1)^n),\ B = n(1 + (-1)^{n+1}).$

Write one solution per line.

Hints and Comments:

- If only one relationship holds, then the most appropriate relationship is just that relationship. If $A = \Theta(B)$ then you should write $A = \Theta(B)$ (even though both $A = O(B)$ and $A = \Omega(B)$ are both also valid). As examples,
  - If $A_1 = 10n$ and $B_1 = n^2$ then the answer should be "$A_1 = O(B_1)$".
  - If $A_2 = 10n^2$ and $B_2 = n^2 - 20n$ then the answer should be "$A_2 = \Theta(B_2)$".
- It is NOT necessary to provide justifications for your answer. But if you do not provide any justifications and you are wrong, we cannot award you partial credit.

Solution:

(a) $A = O(B)$;
Comment: $A = \Theta(n^3)$, $B = \Theta(n^4)$.

(b) $A = \Theta(B)$;

Comment: $B = n \log_2 n$.

From Stirling's formula (taught in first class)

$A = \log_{100}((n{+}100)!) = \Theta((n{+}2)\log(n{+}2)) = \Theta((n{+}2)\log(n{+}2)) = \Theta(n \log n) = \Theta(B)$.

(c) $A = \Omega(B)$;
Comment: $\frac{1}{\log_n 5} = \log_5 n$ so $A = (16)^{\log_5 n} = 2^{4(\log_5 2)\log_2 n}$. Then

$$\frac{A}{B} = 2^{4\log_5 2 \log_2 n - \sqrt{3\log_2 n}} = 2^{\sqrt{\log_2 n}\left(4\log_5 2\sqrt{\log_2 n} - \sqrt{3}\right)} \to \infty.$$

(d) $A = \Theta(B)$;
Comment: $A = \Theta(n^5)$ and $B = \Theta(n^5)$.

(e) $A = \Omega(B)$;
Comment: $A = \Omega(2^n)$. $B = \Theta(n^8)$.

(f) $A = O(B)$
Comment $A = \sum_{k=1}^{n} \frac{1}{k(k+1)} \le \sum_{k=1}^{n} \frac{1}{k^2} < \sum_{k=1}^{\infty} \frac{1}{k^2} < \infty$.

$\sum_{k=1}^{n} \frac{1}{k} = \Theta(\ln n)$ (Harmonic number; taught in first class) so $B = \Theta(\log(\log n))$.

(g) None of the relationships apply.

Comment:

$$A = \begin{cases} 2n & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd} \end{cases}, \qquad B = \begin{cases} 0 & \text{if } n \text{ is even} \\ 2n & \text{if } n \text{ is odd} \end{cases}$$

Suppose $A = O(B)$. Then there exists $c > 0$ and n_0 such that, for all $n \ge n_0$, $A \le cB$.

But, that's impossible because $A(2n_0) = 4n_0 > 0 = cB(2n_0)$.

Similarly, suppose $B = O(A)$. Then there exists $c > 0$ and n_0 such that, for all $n \ge n_0$, $B \le cA$.

But, that's impossible because $B(2n_0+1) = 2(2n_0+1) > 0 = cA(2n_0+1)$.

Problem 3 [20 pts]

Give asymptotic upper bounds for $T(n)$ satisfying the recurrences in parts (a) and (b). Make your bounds as tight as possible.

Note that "tight upper bound" means the best possible Big-O answer. So, $T(n) = O(n^5)$ would not be considered a fully correct answer if $T(n) = O(n^4)$ as well.

You must prove your results from scratch using either the expansion or tree method shown in class. Show all of your work.

For (a) you may assume that n is a power of 3; for (b) you may assume that it is a power of 4.

(a) $T(1) = 1; \quad T(n) = 10T(n/3) + n^2$ for $n > 1$.

(b) $T(1) = 1; \quad T(n) = 16T(n/4) + n^2$ for $n > 1$.

(a) Answer is $T(n) = O\left(n^{\log_3 10}\right)$ or $T(n) = O\left(10^{\log_3 n}\right)$.

Let $h = \log_3 n$ so $n = 3^h$ and $10^h = n^{\log_3 10}$. .

$$\begin{aligned}
T(n) &= 10T\left(\frac{n}{3}\right) + n^2, \quad (\text{for } n > 1)\\
&= 10\left[10T\left(\frac{n}{3^2}\right) + \left(\frac{n}{3}\right)^2\right] + n^2\\
&= 10^2T\left(\frac{n}{3^2}\right) + \left(\frac{10}{3^2} + 1\right)n^2\\
&= 10^2\left[10T\left(\frac{n}{3^3}\right) + \left(\frac{n}{3^2}\right)^2\right] + \left(\frac{10}{3^2} + 1\right)n^2\\
&= 10^3T\left(\frac{n}{3^3}\right) + \left(\frac{10^2}{(3^2)^2} + \frac{10}{3^2} + 1\right)n^2\\
&= \ldots\\
&= 10^iT\left(\frac{n}{3^i}\right) + n^2\left(\sum_{j=0}^{i-1}\left(\frac{10}{3^2}\right)^j\right)\\
&\ldots\\
&= 10^hT\left(\frac{n}{3^h}\right) + n^2\left(\sum_{j=0}^{h-1}\left(\frac{10}{3^2}\right)^j\right).
\end{aligned}$$

Now note that $T\left(\frac{n}{3^h}\right) = T(1) = 1$ and

$$\sum_{j=0}^{h-1}\left(\frac{10}{3^2}\right) = \frac{\left(\frac{10}{3^2}\right)^h - 1}{\frac{10}{3^2} - 1} = 9\left(\frac{10^h}{n^2} - 1\right)$$

so

$$T(n) = 10^hT(1) + n^2 9\left(\frac{10^h}{n^2} - 1\right) = \Theta\left(10^h\right) = \Theta\left(n^{\log_3 10}\right).$$

(b)

Answer is $T(n) = O(n^2 \log n)$.

Let $h = \log_4 n$ so $n = 4^h$. This implies that $16^h = n^2$. Then

$$\begin{aligned}
T(n) &= 16T\left(\frac{n}{4}\right) + n^2, \quad \text{(for } n > 1\text{)} \\
&= 16\left[16T\left(\frac{n}{4^2}\right) + \left(\frac{n}{4}\right)^2\right] + n^2 \\
&= 16^2 T\left(\frac{n}{4^2}\right) + \left(\frac{16}{4^2} + 1\right) n^2 \\
&= 16^2 T\left(\frac{n}{4^2}\right) + 2n^2 \\
&= 16^2\left[16T\left(\frac{n}{4^3}\right) + \frac{n}{(4^2)^2}\right] + 2n^2 \\
&= 16^3 T\left(\frac{n}{4^3}\right) + 3n^2 \\
&= ... \\
&= 16^i T\left(\frac{n}{4^i}\right) + in^2 \\
... \\
&= 16^h T\left(\frac{n}{4^h}\right) + hn^2 \\
&= (T(1) + h)16^h \\
&= (1 + \log_4 n)n^2 \\
&= = \Theta(n^2 \log n).
\end{aligned}$$

Problem 4: [21 pts] Tutorial question DC8 asks you to solve the "stock-profit" problem using divide-and-conquer techniques in $O(n \log n)$ time. The bottom of the first page of the solution slides states:

Note: This problem can be solved using similar ideas that are used for the maximum contiguous subarray problem. Similar to that problem, there is also an $O(n)$ time linear scan solution. That is not what is being requested here, but it is a good exercise to see if you can find it.

You now need to find an $O(n)$ time algorithm for solving the stock-profit problem.

(a) Write your algorithm down in clear documented pseudocode.

Separately, below the algorithm, provide an *understandable* explanation in words as to what the algorithm is doing.

(b) Prove (explain) why your algorithm is correct.

(c) Prove (explain) that your algorithm runs in $O(n)$ time.

Solution:

(a)

Algorithm 1 Stock Profit. Final max profit stored in V.

```
1: $V = 0$;                              ▷ $V$ stores Max-Profit so far
2: Min $= p[1]$;                         ▷ Min stores min value seen so far
3: for $i = 2$ to n do
4:     if $p[i] -$ Min $> V$ then        ▷ Update profit $V$ if appropriate
5:         $V = p[i] -$ Min
6:     if $p[i] <$ Min then              ▷ Update Min value so far if appropriate
7:         Min $= p[i]$
```

- The algorithm scans the array, one item at a time.
- Right before scanning $p[i]$, Min $= \min_{j<i} p[j]$.
- V stores the best profit value found so far.
- If $p[i] -$ Min is greater than the best profit seen so far, then V is updated.

(b) Set $M_1 = 1$, $V_1 = 0$ and for $i > 1$ define

$$M_i = \min_{j<i} p[j]. \quad \text{and} \quad V_i = \max_{j\le i}(p[i] - p[j])$$

The problem is to find V_n.

The major observation is that, for $i > 1$,

$$V_i = \max(V_{i-1}, p[i] - M_i).$$

But this is exactly what the algorithm is calculating.

(c) Each iteration of the for loop on line 3 does $O(1)$ work, so the algorithm runs in $O(n)$ time.

Alternative Solution: There is also a more complicated $O(n)$ time divide and conquer solution.

(a)

Algorithm 2 Stock-Profit(i, j): Returns (m, M, V) where $m = \min_{i \le k \le j} p[k]$, $M = \max_{i \le k \le j} p[k]$ and V is the max possible profit in $p[i \dots j]$.

```
1: if $i = j$ then                                        ▷ $m = p[i]$; $M = p[i]$; $V = 0$
2:     $Return(p[i], p[i], 0)$
3: else
4:     $q = \left\lfloor \frac{i+j}{2} \right\rfloor$
5:     $(m_L, M_L, V_L) = $ Stock-Profit$(i, q)$            ▷ Recursively solve subproblems
6:     $(m_, M_R, V_R) = $ Stock-Profit$(q+1, j)$
7:     $m = \min(m_L, m_R)$; $M = \max(M_L, M_R)$          ▷ Calculate min/max of array
8:     $V = \max(V_L, V_R, M_R - m_L)$
9:     $Return(m, M, V)$
```

This solves the modified problem of finding the (i) profit, (ii) minimum item and (iii) maximum item of an array.

It recursively solves this modified problem on left and right subarrays and uses the results of these two subpbroblems to construct the solution to the modified problem on the full array.

At the same time, it ALSO finds the minimum and maximum items of the left and right sides

(b) The algorithm is obviously correct on subarrays of size one.

Otherwise, it splits the problem on subarray S into problems on left and right subarrays L and R. Let m_X be the minimum item in array X; M_X the maximum item in array X and V_X the maximum profit in array X.

Then, by definition

$$m_S = \min(m_L, m_R), \quad M_S = \max(M_L, M_R).$$

Furthermore, the max profit $V_X = p[i] - p[j]$ is one of the following three cases:

- $i, j \in L$. Then $p[i] - p[j] = V_L$.
- $i, j \in R$. Then $p[i] - p[j] = V_R$.
- $i \in R$ and $j \in L$. This is maximized when $i = M_R$ and $j = m_L$ and then $p[i] - p[j] = M_R - m_L$.

But, line 8 exactly calculates this so it is correct.

(c) The running time of the algorithm satisfies $T(1) = O(1)$ and
$T(n) = 2T(n/2) + O(1)$
which solves (using the Master Theorem) to $T(n) = O(n)$.

Problem 5: [20 pts] **Searching**

The input to the problem is an n element array $A[0, \ldots, n-1]$. Element $A[i]$ is *locally minimal* in array A if $A[i]$ is not larger than all of its neighbors (either the neighbors to both sides, or the neighbor to one side if it's the first or last item). For example, 7, 6 and both 8's are the locally minimal items in the array below

7	**13**	**15**	**12**	**6**	**9**	**11**	**14**	**16**	**23**	**25**	**24**	**8**	**8**

Finding an absolute globally minimal item requires $O(n)$ time to inspect every item. Finding a locally minimal one can be done faster.

You now need to find an $O(\log n)$ time algorithm for finding a locally minimal element in the array. If there are many such items, you only need to return one of them.

(a) Write your algorithm down in clear documented pseudocode.

(b) Prove (explain) why your algorithm is correct.

(c) Prove (explain) that your algorithm runs in $O(\log n)$ time.

Solution:

In order to not have to deal with special cases we will start by **augmenting** our array so that $A[-1] = A[n] = \infty$, where ∞ represents a value larger than any number. As an illustration, we would replace the example on the previous page with

∞	**7**	**13**	**15**	**12**	**6**	**9**	**11**	**14**	**16**	**23**	**25**	**24**	**8**	**8**	∞

This can be implemented in software by, when accessing $A[i]$, having the algorithm check if $i = -1$ or or $i = n + 1$ and, if yes, returning that those items are larger than anything to which they are compared.

The important observation is that **the local minima of the original array are exactly the same as the local minima of the augmented array.**

It is not absolutely necessary to create this augmented array but doing so removes special cases in the correctness proof for the algorithm that would otherwise have to be dealt with differently.

The algorithm is now going to work by maintaining a subarray (halving the size each time). At completion, when the subarray reaches size 1 or 2, we will see that it must contain a local minimum.

In what follows,
Global Local Minimum (GLM) refers to a local minimum of $A[0 \dots n]$;
Local Minimum of $A[i \dots j]$ refers to a local minimum of the $A[i \dots j]$ subarray.

Note that, a-priori, a **Local Minimum of** $A[i \dots j]$ does NOT have to be a **Global Local Minimum**.

(a) Our algorithm, $LM(i, k)$ will be initially called as $LM(0, n - 1)$.

Algorithm 3 LM(i, k): Assumes $A[i-1] \geq A[i]$ and $A[k] \leq A[k+1]$. Returns index j of Global Local Minimum located within $A[i \dots k]$

```
1: if $k = i$ then                                     ▷ Size 1 problem
2:     $Return(i)$
3: else if $k = i + 1$ then                            ▷ Size 2 problem
4:     if $A[i] \leq A[i+1]$ then
5:         $Return(i)$
6:     else
7:         $Return(k)$
8: else                                                ▷ Size > 2 problem
9:     $j = \lfloor \frac{i+k}{2} \rfloor$
10:    if $A[j-1] < A[j]$ then             ▷ $A[i-1] \geq A[i]$ and $A[j-1] \leq A[j]$
11:        $Return(LM(i, j-1))$
12:    else                                ▷ $A[j-1] \geq A[i]$ and $A[k] \leq A[k+1]$
13:        $Return(LM(j, k))$
```

A call of $LM(i,k)$ will assume that

$$(*) \quad A[i-1] \geq A[i] \quad \text{and} \quad A[k] < A[k+1].$$

Note that the initial call $LM(0, n-1)$ satisfies $(*)$.

Marking Note: 5 (a)

This was not the only solution. It was also possible to write code that explicitly checked whether $A[j]$, the midpoint of the subarray, was a local minimum and, if not, recursing left or right appropriately.

All correct algorithms would have to recurse into smaller subarrays, finding the local minimum of the subarrays. The major issue in proving correctness is in proving that the local minimum of smaller subarray $A[i \ldots j]$ is a **global local minimum**.

That statement is not true though. In the example subarray the local minimum of $A[1 \ldots 3] = [1,\ 15,\ 12]$ is 12, but 12 is not a **global local minimum** (because its neighbor 4 is smaller than it in the full array).

What is true is the more restricted lemma below.

Lemma: If $A[j]$ is a local minimum in $A[i \ldots k]$, and (*) is satisfied, then $A[j]$ is a global local minimum (GLM).

Proof: Suppose $A[j]$ is a local minimum in $A[i \ldots j]$, and (*) is satisfied.
Then $A[i-1] \geq A[i]$ and $A[k] < A[k+1]$.

There are four possible cases for j.

- $i < j < k$: Then $A[j-1] \geq A[j] \leq A[j+1]$ so $A[j]$ is a GLM.
- $i = j < k$: Since $A[j]$ is a local minimum, $A[i=j] \leq A[j+1]$.
  Since $A[i-1] \geq A[i=j]$ from (*), $A[j]$ is a GLM.
- $i < j = k$: Since $A[j]$ is a local minimum, $A[j=k] \leq A[k+1]$
  Since $A[k-1] \geq A[j=k]$ from (*), $A[j]$ is a GLM.
- $i = j = k$: Then, from (*), $A[i-1] \geq A[i=j=k] \leq A[k+1]$
  so $A[j]$ is a GLM.

■

(b) Correctness:

First note that the algorithm always terminates, returning a value.

If the size $k - i + 1 \leq 2$ the algorithm terminates. If the size $k - i + 1 > 2$ then, at the next subcall, the size decreases. So, the algorithm must terminate.

To prove correctness we therefore only need to show that when the algorithm returns a value in steps 2, 5 or 7, the value returned is a **global local minimum**.

We do this in two parts.

(i) We first show that the algorithm always maintains the correctness of $(*)$.

(ii) We next show that *because* $(*)$ *is correct at termination, the value returned is a global local minimum.*

(i) We prove by induction that invariant $(*)$ is always maintained. Invariant $(*)$ is correct in the first step (by the array augmentation). We next show that if steps 11 or 13 are invoked, then the recursively called subarray satisfies $(*)$.

- If step 11 is invoked, then $LM[i, j-1]$ is called.
  From $(*)$ on the current step, $A[i-1] \geq A[i]$.
  Step 11 is only invoked when $A[j-1] < A[j]$, so $A[i, j-1]$ also satisfies $(*)$.

- If step 13 is invoked, then $LM[j, k]$ is called.
  From $(*)$ on the current step, $A[k] \leq A[k+1]$.
  Step 13 is only invoked when $A[j-1] \geq A[j]$, so $A[j, k]$ also satisfies $(*)$.

Thus $(*)$ is always satisfied.

(ii) We now show that when the algorithm returns value j that $A[j]$ must be a global local minimum.

Note that lines 2, 5 and 7 are always returning the local minimum of the size 1 or 2 array $A[i \ldots k]$. Since (*) is correct, by the Lemma, the returned value is also a GLM.

Comment.

The Lemma actually was stronger than needed. The first case in the lemma would actually never occur in our algorithm because all of our termination arrays have size 1 or 2. We included this extra case to use it in the marking note below.

Marking Note: 5 (b)

A valid proof of correctness must have shown that, at termination, the value returned is a GLOBAL LOCAL MINIMUM not just a local minimum of the subarray on which the procedure was called.

This required either explicitly or implicitly proving that when the algorithm recursed, the side recursed into contained a local minimum., i.e, proving something similar to the Lemma.

Proofs of correctness missing this step were marked wrong and had points deducted.

Going Further. Many students tried to prove correctness by using an induction hypothesis of the form

"$I(m)$: The algorithm correctly finds a local minimum in every subarray $A[i \ldots j]$ with size $m = j - i + 1$."

Any proof trying this would be incorrect. T Such a hypothesis implicitly assumed that a local minimum in $A[i \ldots j]$ is a global local minimum and this is not always true.

A correct induction would use a hypothesis more like

"$I(m)$: If subarray $A[i \ldots j]$ with size $m = j - i + 1$ contains a global local minimum then the algorithm run on subarray $A[i \ldots j]$ finds such a global local minimum."

Proving this requires more work, e.g., using something like condition (*).

Going Even Further. The major issue was that some students confused the two very different concepts of local minimum of subarray $A[i \ldots k]$ and of Global Local minimum.

If those two were not kept separate in the proof of correctness it was very easy to get the proof wrong.

(c) this algorithm is halving the size of the subarray at every step so it only uses $O(\log n)$ steps. Each step does $O(1)$ work so entire algorithm does only $O(\log n)$ work.