Problem 1

(a) (i) For all $k > 0$, we have $T\left(3^k\right) \leq T\left(3^{k-1}\right) + c \leq T\left(3^{k-2}\right) + 2c \leq \cdots \leq T\left(3^{k-k}\right) + kc = 1 + kc$. Hence, $T\left(3^k\right) \leq 1 + kc = O(k)$. (The constant 1 is discarded since it becomes insignificant for large n.)

(ii) We start with $n = 3^{\log_3 n} \leq 3^{\lceil \log_3 n \rceil}$. [1]

$$
\begin{aligned}
S(n) &\leq S(3^{\lceil \log_3 n \rceil}) && \text{since } S \text{ is non-decreasing} \\
&= O(\lceil \log_3 n \rceil) \\
&\leq c\lceil \log_3 n \rceil \\
&\leq c(1 + \log_3 n) \\
&\leq 2c \log_3 n && \text{since } n \geq 3 \\
&= O(\log_3 n)
\end{aligned}
$$

(iii) $R(n) = T(i)$ for some $1 \leq i \leq n$. Then $R(n) \leq T\left(\left\lfloor \frac{i}{3} \right\rfloor\right) + c$. Since $1 \leq \left\lfloor \frac{i}{3} \right\rfloor \leq \left\lfloor \frac{n}{3} \right\rfloor$, so $T\left(\left\lfloor \frac{i}{3} \right\rfloor\right) \leq \max_{1 \leq \left\lfloor \frac{j}{3} \right\rfloor \leq \left\lfloor \frac{n}{3} \right\rfloor} T\left(\left\lfloor \frac{j}{3} \right\rfloor\right)$ because $T\left(\left\lfloor \frac{i}{3} \right\rfloor\right)$ is either the maximum or there is some other maximum in that range (between 1 and $\left\lfloor \frac{n}{3} \right\rfloor$) greater than $T\left(\left\lfloor \frac{i}{3} \right\rfloor\right)$. Hence, $R(n) \leq \max_{1 \leq \left\lfloor \frac{j}{3} \right\rfloor \leq \left\lfloor \frac{n}{3} \right\rfloor} = R\left(\left\lfloor \frac{n}{3} \right\rfloor\right) \leq R\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c$.

(iv) We first show that $R(3^k) = O(k)$. By definition, $R(1) = R(2) + 1$ and $\forall n > 2$, $R(n) \leq R\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c$. It thus follows from (i) that $R(3^k) = O(k)$.

Now note that R is non-decreasing, since at each point, we're taking the maximum of the previous values and a new value $T(n)$. Inductively, this is implied by

$$R(n) = \max(R(n-1), T(n)) \geq R(n-1).$$

From (ii), since $R(n)$ is non-decreasing over n and $R(n)$ satisfies $R(3^k) = O(k)$, it

[1] Proof derivation assisted by *redacted*.

follows that $R(n) = O(\log_3 n)$. This result means $R(n) \leq c \log_3 n$ for some constant $c > 0$ and for large enough n.

Finally,

$$\begin{aligned} T(n) &\leq \max_{1 \leq i \leq n} T(i) \\ &\leq R(n) \\ &\leq c \log_3 n \\ \implies T(n) &= O(\log_3 n) \end{aligned}$$

(b) (v) Similar to (ii), we have $n = 2^{\log_2 n} \leq 2^{\lceil \log_2 n \rceil}$.

$$\begin{aligned} S(n) &\leq S\left(2^{\lceil \log_2 n \rceil}\right) && \text{since } S \text{ is non-decreasing} \\ &= O\left(\lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil}\right) \\ &= c \lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil} \\ &\leq c(1 + \log_2 n) 2^{1 + \log_2 n} \\ &= 2c(1 + \log_2 n) n \\ &\leq 2c(2 \log_2 n) n && \text{since } n \geq 2 \\ &= O(n \log_2 n) \end{aligned}$$

(vi) Similar to (iii). For all $n \geq 2$, we have $R(n) = T(i)$ for some $1 \leq i \leq n$. Since $1 \leq \left\lfloor \frac{i}{2} \right\rfloor \leq \left\lfloor \frac{n}{2} \right\rfloor$ and $1 \leq \left\lceil \frac{i}{2} \right\rceil \leq \left\lceil \frac{n}{2} \right\rceil$ and hence,

$$\begin{aligned} R(n) &= T(i) \\ &\leq T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \\ &\leq \max_{1 \leq \left\lfloor \frac{j}{2} \right\rfloor \leq \left\lfloor \frac{n}{2} \right\rfloor} T\left(\left\lfloor \frac{j}{2} \right\rfloor\right) + \max_{1 \leq \left\lceil \frac{j}{2} \right\rceil \leq \left\lceil \frac{n}{2} \right\rceil} T\left(\left\lceil \frac{j}{2} \right\rceil\right) \\ &= R\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + R\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \end{aligned}$$

(vii) Taking a similar approach to (iv), we apply (v) and (vi) to show that $T(n) = O(n \log_2 n)$.

$R(n)$ is non-decreasing over n for the same reason that $R(n)$ in (iii) is non-decreasing (proven in (iv)). By application of (v) that $R(n) = O(n \log_2 n)$.

It follows that

$$\begin{aligned} T(n) &\leq \max_{1 \leq i \leq n} T(i) \\ &= R(n) \\ &= O(n \log_2 n) \end{aligned}$$

Problem 2

(a) $A = O(B)$. B grows faster for large n.

(b) $A = O(B)$. n^n grows faster than $n!$, and even $(n+2)!$ since only a polynomial factor is introduced. The logarithmic base is irrelevant thanks to change of bases.

(c) $A = \Omega(B)$. $A = 2^{4\log_5 n}$, $B = 2^{\sqrt{3\log_2 n}}$. A grows faster than B for large n since $4 > 3$ and the logarithm is exponentiated.

(d) $A = \Theta(B)$. Both A and B are fifth-degree polynomials.

(e) $A = \Omega(B)$. A (dominated by exponential) grows faster than B (polynomial) for large n.

(f) $A = O(B)$. $A = \frac{n}{n+1}$ and converges to 1. $B = \ln H_n$ where H_n is the n-th harmonic series and grows logarithmically. Since B diverges to infinity, B grows faster than A.

(g) None of the relations are satisfied.

Problem 3

(a) Assume $n = 3^k$ (also $k = \log_3 n$) for non-negative integers k. Expanding $T(n)$...

$$
\begin{aligned}
T\left(3^k\right) &= 10T\left(3^{k-1}\right) + \left(3^k\right)^2 \\
&= 10\left(10T\left(3^{k-2}\right) + \left(3^{k-1}\right)^2\right) + \left(3^k\right)^2 \\
&= 10^2 T\left(3^{k-2}\right) + 10\left(3^{k-1}\right)^2 + \left(3^k\right)^2 \\
&= 10^i T\left(3^{k-i}\right) + 10^{i-1}\left(3^{k-i+1}\right)^2 + \cdots + 10\left(3^{k-1}\right)^2 + \left(3^k\right)^2 \\
&= 10^k + 10^{k-1}\left(3^1\right)^2 + \cdots + 10\left(3^{k-1}\right)^2 + \left(3^k\right)^2 \\
&= \sum_{i=0}^{k} 10^{k-i}\left(3^i\right)^2 \\
&= 10^k \sum_{i=0}^{k} \left(\frac{9}{10}\right)^i \\
&= 10^k \frac{1 - \left(\frac{9}{10}\right)^{k+1}}{1 - \frac{9}{10}} \\
&= 10^{k+1}\left(1 - \left(\frac{9}{10}\right)^{k+1}\right) \\
&= 10^{k+1} - 9^{k+1} \\
&= 10^{(\log_3 n)+1} - 9^{(\log_3 n)+1} \\
&= 10^{\log_3 3n} - 9^{\log_3 3n} \\
&= \left(3^{\log_3 10}\right)^{\log_3 3n} - \left(3^2\right)^{\log_3 3n} \\
&= (3n)^{\log_3 10} - (3n)^2 \\
&= O(n^{\log_3 10})
\end{aligned}
$$

(b) Assume $n = 4^k$ (also $k = \log_4 n$) for non-negative integers k. Expanding $T(n)$...

$$\begin{aligned}
T\left(4^k\right) &= 16T\left(4^{k-1}\right) + \left(4^k\right)^2 \\
&= 16^k + 16^{k-1}\left(4^1\right)^2 + \cdots + 16\left(4^{k-1}\right)^2 + \left(4^k\right)^2 \\
&= \sum_{i=0}^{k} 16^{k-i}\left(4^i\right)^2 \\
&= \sum_{i=0}^{k} 16^k \\
&= (k+1)16^k \\
&= ((\log_4 n)+1)16^{\log_4 n} \\
&= ((\log_4 n)+1)n^2 \\
&= ((\log_4 n)+1)n^2 \\
&= n^2 \log_4 n + n^2 \\
&= O(n^2 \log_4 n)
\end{aligned}$$

Problem 4

(a) Algorithm[2]:

Algorithm 1 StockProfitSolver

```
1:  function StockProfitSolver(A, n)
2:      V_max ← −∞, X_min ← 0, X ← 0, V ← 0
3:      i_l ← −1, i_lbest ← −1, i_rbest ← −1
4:      for i ← 1 to n do
5:          X ← X + A[i]
6:          V ← V + A[i]
7:          if i_l = −1 then
8:              i_l ← i                          ▷ Start a new span if not in one
9:          end if
10:         if V > V_max then                    ▷ Check if we've hit a new high
11:             V_max ← V                        ▷ Update current best sum to V
12:             i_lbest ← i_l
13:             i_rbest ← i                      ▷ Update current best span to (i_l, i)
14:         end if
15:         if X < X_min then                    ▷ This is a new low...
16:             X_min ← X                        ▷ Update the new low
17:             V ← 0                            ▷ Reset V and i_l
18:             i_l ← −1
19:         end if
20:     end for
21:     if V_max < 0 then                        ▷ Best profit is a loss
22:         return "no way"                      ▷ :(
23:     end if
24:     return (i_lbest, i_rbest)
25: end function
```

This algorithm makes one pass through array A. It keeps track of the best high (V_{max}), moving high (V), global low (X_{min}), incremental sum (X), best left index (i_{lbest}), temporary left index (i_l), and best right index (i_{rbest}).

For each element, it first updates the moving high (V) and incremental sum (X) by adding the current element to both.

It then initialises the temporary left index i_l, if it hasn't already been initialised. We use -1 to indicate that the index is invalid and uninitialised. A valid index would be between

[2]Adapted from the $O(n)$ pseudocode from the max subarray lecture notes, with modifications for the stock-profit problem.

1 and n inclusive.

We then check to see if the new V is better than supposed "best high" V_{max}, and if so, perform an update to V_max. Here, we also update the best pair i_{lbest}, i_{rbest}. This is the pair we will be returning at the end of the function. We update the left bound i_{lbest} with the temporary index i_l. (Note that i_l would be a valid index at this point since it's already been initialised (either on this iteration or some iteration before).) We also update the right bound to the current index i, since the new V was updated on the same iteration.

Next we check if a new global low has been reached ($X < X_{min}$). If true, then we could find a better solution by resetting, instead of carrying on with the current state. So we reset V to 0, i_l to -1, and update X_{min} with X.

(b) We already know that the max subarray problem could be correctly solved using lines 2, 4, 5, 6, 10, 11, 14, 15, 16, 17, 19, and 20. The idea that is carried across is of the sum of the subarray, and the new additions are the variables storing the left and right indices of the best subarray.

We know that $i_l \leq i$ (or $i_l = -1$ if invalid), since i_l is set to i in line 8. In lines 12-13, we set i_{lbest} to i_l and i_{rbest} to i, so naturally we have $i_{lbest} \leq i_{rbest}$; and this satisfies the constraint set by the problem.

(c) The algorithm makes one pass through the for-loop. The comparisons and assignments in each iteration can be bounded by a constant $c > 0$. For large enough n (the size of A), this constant becomes insignificant. Thus, we have $T(n) + c = O(n)$.

Problem 5

(a) Algorithm for finding a local minimum[3]:

Algorithm 2 Local Minimum Search

```
1:  function LocalMinimumSearch(A, l, r, n)
2:      m ← ⌊(l+r)/2⌋
3:      if (m = 0 or A[m−1] ≥ A[m])                ▷ A[m] is locally minimal from left
4:          and (m = n−1 or A[m] ≤ A[m+1]) then    ▷ A[m] is locally minimal from right
5:          return m                                ▷ Terminate and return index
6:      else if m > 0 and A[m−1] < A[m] then
7:          return LocalMinimumSearch(A, l, m−1, n)   ▷ Recurse on left subarray
8:      else                                        ▷ (m < n−1 and A[m] > A[m+1])
9:          return LocalMinimumSearch(A, m+1, r, n)   ▷ Recurse on right subarray
10:     end if
11: end function
```

First call: LocalMinimumSearch(A, 0, $n-1$, n), where n is the length A.

(b) Let s be the number of elements considered in the "current" search, $A[l..r]$, with $s = r - l + 1$. We will prove the correctness of the algorithm by induction over s.

$I(s)$: For every array A and all $l \leq r \leq n$ with $r - l + 1 = s$, then LocalMinimumSearch(A, l, r, n) returns $i \in [l..r]$ such that $A[i]$ is locally minimal.

$A[i]$ is locally minimal if any of the following are true:

1. $i = 0$ and $i = n - 1$
2. $i = 0$ and $A[i] \leq A[i+1]$
3. $i = n - 1$ and $A[i-1] \geq A[i]$
4. $A[i-1] \geq A[i]$ and $A[i] \leq A[i+1]$

Base Case ($s = 1$ or $A[m]$ is locally minimal). If $n = 1$, then we have $m = l = r = 0 = n - 1$. This fulfills the first type of local minimum. and the function correctly returns 0.

For $n > 1$, we need to consider the three other cases where $A[m]$ is a local minimum. By

[3]Algorithm sourced from `https://www.geeksforgeeks.org/find-local-minima-array/`. Note that the pseudocode shown here uses $\leq$ and $\geq$ in the first condition (lines 3-4) whereas the code in GeeksforGeeks uses strict inequalities.

assumption $A[m]$ is a local minimum, and Algorithm 2 handles this and correctly returns m.

General Case ($s > 1$ and $A[m]$ is not locally minimal). We assume $I(s)$ to be true for smaller s.

There are two cases to consider here:

1. $A[m-1] < A[m]$, in which case a local minimum exists[4] on the left subarray $A[l..m-1]$. We perform recursion and return the result from LOCALMINIMUMSEARCH(A, l, $m-1$, n). Since this is a subproblem with smaller s ($= m-l$), the correct answer is returned.
2. $A[m] > A[m+1]$, in which case a local minimum exists[4] on the right subarray $A[m+1..r]$. Similar to the first case, we return the result from the subproblem LOCALMINIMUMSEARCH(A, $m+1$, r n). Since s is smaller, the correct answer is returned.

Hence, $I(s)$ is true for all s.

(c) We have $T(1) = 1$ and $\forall s > 1$. $T(s) \leq T\left(\left\lfloor \frac{s}{2} \right\rfloor\right) + c$, where $c > 0$ is a constant denoting an arbitrary number of comparisons, bounded by $O(1)$. An inequality was used for the general $T(s)$ case since there is the possibility that the local minimum is found on the first iteration or on the last iteration.

By the master theorem, since we have $c = \log_2 1 = 0$ and $f(s) = O(1)$, thus $T(s) = O(\log s)$.

[4]We can show that a local minimum exists in the subarray $A[l..m-1]$ using induction (again). We first assume that a local minimum exists in $A[l..r]$. Now after splitting A, if $A[l..m-1]$ contains only one element, then since $A[m-1] < A[m]$, the local minimum exists and is $A[m-1]$.

We assume then, that since $A[m-1]$ isn't a local minimum, then $A[m-2] < A[m-1]$. And we inductively test $A[m-i]$ down until $A[l]$ if necessary. Thus a local minimum exists. The same reasoning goes for the right subarray $A[m+1..r]$.