Q1

Algorithm:

Let n the size of A

```
LocalMin(A):
1	if A[1] < A[2]:
2		return A[1]
3	else if A[n] < A[n - 1]:
4		return A[n]
5	else:
6		mid = ceiling( n/2 )
7		if (A[mid - 1] > A[mid] < A[mid + 1]):
8			return A[mid]
9		else if (A[mid - 1] < A[mid] < A[mid + 1]):
10			return LocalMin(A[1: mid])
11		else:
12			return LocalMin(A[mid:n])
```

Proof of Correctness:

By Complete Induction,

Basis: n = 2

If A[1] < A[2] ⇒ A[1] is a local minimum

By line 1-2, A[1] get returned, as wanted

If A[1] > A[2] ⇒ A[2] is a local minimum

Since n = 2, A[n] = A[2] < A[1] = A[n-1]

By line 3-4, A[2] get returned, as wanted

IH:

Suppose for any A with size k < n, this algorithm can find a local minimum

IS:

WTS: for A with size n, this algorithm will also find a local minimum

There are 4 cases:

1. A[mid-1] > A[mid] < A[mid+1]

   ⇒ A[mid] itself is a local minimum

   By line 7-8, A[mid] a local minimum get returned.

2. A[mid-1] < A[mid] < A[mid+1]

   ⇒ there will be at least one local minimum at left part i.e. within A[1:mid]
   Aside: A[1] > A[2] … A[mid - 1] < A[mid]
   By line 9-10 and IH, LocalMin(A[1: mid]) will produce a local minimum which will get returned.

3. A[mid-1] > A[mid] > A[mid+1]

   ⇒ there will be at least one local minimum at right part i.e. within A[mid:n]
   Aside: A[mid] > A[mid+1] … A[n-1] < A[n]

   By line 11-12 and IH, LocalMin(A[mid: n]) will produce a local minimum which will get returned.

4. A[mid-1] < A[mid] > A[mid+1]

   ⇒ both left part and right part will contain a local minimum
   Aside: A[1] > A[2] … A[mid-1] < A[mid] > A[mid+1] … A[n-1] < A[n]

   By line 11-12 and IH, LocalMin(A[mid: n]) will produce a local minimum which will get returned.

   Therefore this algorithm will produce a local minimum for an array of any size

Running Time:

$T(n) = T(\frac{n}{2}) + C$

$a = 1, b = 2, d = 0 \quad \Rightarrow a = b^d$

$T(n) = \Theta(n^d \log n) = \Theta(\log n)$ by master thm

Q-2

Algorithm:

Let n be the size of A

```
Sum(A):  ⇐ helper function
i = 1
S = [0] * n                    (initialize S be an array of size n containing all 0s)
while (i <= n):
        if (i == 1):
                S[i] = A[i]
        else:
                S[i] = A[i] + S[i - 1]
        i = i + 1
return S
```

S = Sum(A) ⇐ Put the helper function call here, so that we can directly use S[j] -S[i] below

MaxValue(A):

```
        if n == 1:
                return A[1]
        else:
                left = MaxValue(A[ 1, … , n/2 ])
                right = MaxValue(A[ n/2 +1, ... , n ])

                start = infinite
                end = -infinite
                for i = 1,..., n/2 :
                        if (start > S[i]):
                                start = S[i]
                for j =  n/2 +1,…,n:
                        if (end < S[j]):
                                end = S[j]
                mid = end - start
                return max{left, right, mid}
```

Proof of Correctness:

By Complete Induction,

Basis: Let n = 1, which means that there is only one integer in the list.

The maximum value of subarray of A is the only integer A[1].

Since length of A is 1, line 1 is satisfied, so executes line 2, which returns A[1], as wanted.

Induction Hypothesis:

A is an array of length k.

MaxVal(A) would output the maximum value of the subarray of A, where $1 \leq k < n$ and assume n is a power of 2.

Induction Steps: Let A have a size of n is a power of 2.

W.T.S: MaxValue(A) would output the maximum value of the subarray of A

Since n > 1, line 4-16 is executed.

$\text{len}(A[\,1 \ldots \frac{n}{2}\,]) = \text{len}(A[\,\frac{n}{2} + 1 \ldots n\,]) = \frac{n}{2} < n$

⇒ *left* has the maximum value of the subarray in the first half of A[By IH], and *right* has the maximum value of the subarray in the last half of A [By IH]

By line 6-15, *mid* is the max value of the subarray of A that containing both A[$\frac{n}{2}$] and A[$\frac{n}{2}$ + 1] (i.e. the two mid points of the array).

In addition, max{*left*, *right*} would find the maximum value of the subarray that are not both containing A[$\frac{n}{2}$] and A[$\frac{n}{2}$ + 1] (i.e. the mid point of the array).

Therefore, max{*left, right, mid*} would find the maximum value of A cover above two cases.

Running Time:

$T(n) = T(n/2) + n + C$ note: construct S and find *mid* use O(n) time

$a = 2, b = 2, d = 1 \Rightarrow a = b^d$

$T(n) = \Theta(n^d \log n) = \Theta(n \log n)$ by master thm

Q-3-a:

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix}$$

Q-3-b

Consider M:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \end{bmatrix}$$

such that $M\underline{a}^T = \underline{v}^T$, where $\underline{a} = (a_0, a_1, a_2, a_3, a_4)$ and $\underline{v} = (p(0), p(1), p(2), p(3), p(4))$

$\Rightarrow (M^{-1}M)\underline{a}^T = M^{-1}\underline{v}^T$
$\Rightarrow \underline{a}^T = M^{-1}\underline{v}^T$
$\Rightarrow M_2 = M^{-1}$
(source:http://www.math.odu.edu/~bogacki/cgi-bin/lat.cgi)

M_2:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{25}{12} & 4 & -3 & \frac{4}{3} & -\frac{1}{4} \\ \frac{35}{24} & -\frac{13}{3} & \frac{19}{4} & -\frac{7}{3} & \frac{11}{24} \\ -\frac{5}{12} & \frac{3}{2} & -2 & \frac{7}{6} & -\frac{1}{4} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \end{bmatrix}$$

Q-3-c

Algorithm:

MULT(A, B):
1 if n == 1 then
2 return A[0]B[0]
3 else
4 t = n / 3
5 a_0 = A[0:t]; a_1 = A[t:2t]; a_2 = A[2t:n]
6 b_0 = B[0:t]; b_1 = B[t:2t]; b_2 = B[2t:n]

Evaluation
7 $[A_0, A_1, A_2, A_3, A_4] = M_1[a_0, a_1, a_2]$
8 $[B_0, B_1, B_2, B_3, B_4] = M_1[b_0, b_1, b_2]$

9 for i = 0 to 4 do
10 C_i = MULTI(A_i, B_i)

interpolation
11 $[c_0, c_1, c_2, c_3, c_4] = M_2[C_0, C_1, C_2, C_3, C_4]$
12 return $[c_4, c_3, c_2, c_1, c_0]$

Proof of Correctness:

Basis: n = 1

Simply multiply two 1-bit numbers, A[0]B[0] get returned as wanted

Induction Hypothesis:

Suppose this algorithm will return a multiplication of any two bit-numbers with length k, where 1 <= k < n

Induction Steps:

WTS, MULI(A,B) will produce a multiplication of A and B where A and B have length n

By line 5-6, we know a_i and b_i are coefficient repenstations of P_a and P_b:

$A = P_a(2^{n/3}) = [a_2, a_1, a_0]$ where $P_a(x)= a_2x^2 + a_1x + a_0$

$B = P_b(2^{n/3}) = [b_2, b_1, b_0]$ where $P_b(x)= b_2x^2 + b_1x + b_0$

We want to compute C = AB by computing $P_c(2^{n/3})$, so we need P_c

Since P_a and P_b are polynomial of degree 2, $P_c = P_aP_b$ is a degree of 4.

Therefore we need 5 points to interpolate P_c. Let x = 0, 1, 2, 3, 4.

By line 7-8 and part a, we can evaluate below value representations of P_a and P_b:

$[A_0, A_1, A_2, A_3, A_4] = [P_a(0), P_a(1), P_a(2), P_a(3), P_a(4)] = M_1[a_0, a_1, a_2]$
$[B_0, B_1, B_2, B_3, B_4] = [P_b(0), P_b(1), P_b(2), P_b(3), P_b(4)] = M_1[b_0, b_1, b_2]$

$A_i = (M_1)_{i+1,\,1} * a_0 + (M_1)_{i+1,\,2} * a_1 + (M_1)_{i+1,\,3} * a_2 \qquad i = 0,1,2,3,4$
$B_i = (M_1)_{i+1,\,1} * b_0 + (M_1)_{i+1,\,2} * b_1 + (M_1)_{i+1,\,3} * a_2 \qquad i = 0,1,2,3,4$
(This takes linear time, since multiplication take constant time since elements in M_1 are small and adding 3 bit number with length n/3 take linear time)

$\Rightarrow A_i$ or B_i have length n/3 $\qquad i = 0,1,2,3,4$

By line 9-10 and IH (since A_i and B_i have length n/3 < n), we can get value representations of P_c (i.e. C_i i = 0, 1, 2, 3, 4):
$[C_0, C_1, C_2, C_3, C_4] = [MULT(A_0, B_0), MULT(A_1, B_1), MULT(A_2, B_2), MULT(A_3, B_3), MULT(A_4, B_4)]$

By line 11 and part2, we can get the coefficient representations of P_c:
$[c_0, c_1, c_2, c_3, c_4] = M_2[C_0, C_1, C_2, C_3, C_4]$
$\Rightarrow P_c(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$

$\Rightarrow P_c(2^{n/3}) = [c_4, c_3, c_2, c_1, c_0]$ is a multiplication of A and B where A and B have lengths of n

By line 12, $[c_4, c_3, c_2, c_1, c_0]$ get returned as wanted.

Therefore this algorithm can return a multiplication of any two bit-numbers with an arbitrary length.

Running time:
$T(n) = 5T(n/3) + Cn$
$a = 5, b = 3, d = 1 \Rightarrow a > b^d$
$T(n) = \Theta(n^{\log 3(5)}) = \Theta(n^{1.46497...})$ by master thm

Karatsuba's algorithm has running time: $T(n) = \Theta(n^{\log 2(3)}) = \Theta(n^{1.585...})$

Therefore faster than Karatsuba's algorithm **(**$\Theta(n^{1.46497...}) < \Theta(n^{1.585...})$**)**