Student who wrote the solution to this question: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai
Students who read this solution to verify its clarity and correctness: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai

2. (a) i. Divide A into two subarrays L, R of equal size at the midpoint. (since n is a power of 2, we treat 'midpoint' as either the tail of the left subarray or the head of the right subarray)
   ii. Find the maximum subarray sum of the left half and the maximum subarray sum of the right half recursively.
   iii. Find the maximum subarray sum of A that cross the midpoint, which means that this subarray starts in the left subarray and ends in the right subarray. This can be done in linear time because we only need to find the maximum subarray sum that end with the tail of the left subarray L_{tail} and the maximum subarray sum that start with the head of the right subarray R_{head}, then the maximum subarray sum of A that cross the midpoint is $L_{tail} + R_{head}$.
   iv. Then the maximum subarray of A is the maximum of the above three cases, because they cover all possible maximum subarrays of A (one that only contain the left half, one that only contain the right half, and the one that crosses the midpoint).

   Pseudocode:

```
# start the algrithm by calling max_subarray(A, 1, n), n is the length of A

 max_subarray(A, i, j): # main body of the algorithm
    if i == j:
        return A[i]
    mid = (i + j) // 2      # mid is the index of the last ele of left subarray
    L_max = max_subarray(A, i, mid)
    R_max = max_subarray(A, mid + 1, j)
    M_max = max_subarray_mid(A, i, mid, j)
    return MAX(L_max, R_max, M_max)

 max_subarray_mid(A, i, mid, j): # find the maximum subarray that cross mid
    L_tail = A[mid]
    sum = 0
    for r in A[mid:i - 1:-1]:
        sum += r
        if sum > L_tail:
            L_tail = sum
    R_head = A[mid + 1]
    sum = 0
    for r in A[mid + 1:j]:
        sum += r
        if sum > R_head:
            R_head = sum
    return L_tail + R_head
```

   The recurrence relation is $T(n) = 2 \cdot T(n/2) + c \cdot n$ because the combine step iterate through the array from the midpoint which takes linear time. By the Master Theorem, this algorithm runs in $O(n \log n)$.

(b) i. Divide A into two subarrays L, R of equal size at the midpoint.

ii. For each of the left and right subarray, find the following recursively:

A. Total sum of the subarray, L_{total}, R_{total}.

B. Maximum subarray sum that start with the head of the subarray, L_{head}, R_{head}.

C. Maximum subarray sum that end with the tail of the subarray, L_{tail}, R_{tail}.

D. Maximum subarray sum of the subarray, L_{max}, R_{max}.

iii. Then we can combine the two subproblems in the following way:

A. Total sum of A is the sum of the total sums of two subarrays, so $A_{total} = L_{total} + R_{total}$.

B. Maximum subarray sum that start with the head of A, A_{head}. If $L_{head} < L_{total} + R_{head}$) then A_{head} cross the midpoint, otherwise it doesn't cross midpoint. So, $A_{head} = \max(L_{head}, L_{total} + R_{head})$.

C. Similarly, $A_{tail} = \max(R_{tail}, L_{tail} + R_{total})$.

D. Just like in part (a), the maximum subarray sum of A is the maximum of the three cases: $A_{max} = \max(L_{max}, R_{max}, L_{tail} + R_{head})$.

A_{max} is the final result we are looking for.

Pseudocode:

```
# start the algrithm by calling max_subarray(A, 1, n), n is the length of A.
# the last ele of the return value is the result we are looking for.

max_subarray(A, i, j):
    if i == j:
        return (A[i], A[i], A[i], A[i])
    else:
        mid = (i + j) // 2
        L_total, L_head, L_tail, L_max = max_subarray(A, i, mid)
        R_total, R_head, R_tail, R_max = max_subarray(A, mid + 1, j)
        A_total = L_total + R_total
        A_head = MAX(L_head, L_total + R_head)
        A_tail = MAX(R_tail, L_tail + R_total)
        A_max = MAX(L_max, R_max, L_tail + R_head)
        return (A_total, A_head, A_tail, A_max)
        # A_max is the maximum subarray sum of A
```

The recurrence relation is $T(n) = 2 \cdot T(n/2) + c$ because the combine step now takes constant time. By the Master Theorem, this algorithm runs in $O(n)$.