Final_Project

組別：G。B0521229 林威廷、B0521233 陳崇瀚

1. Use the divide-and-conquer approach to write a recursive program that finds the maximum sum in any contiguous sublist of a given list of n real values. Analyze your algorithm, and show the results in order function.

*Test Result *

(1) array[] = { -24, 0, 66, -12, 66, 1 };

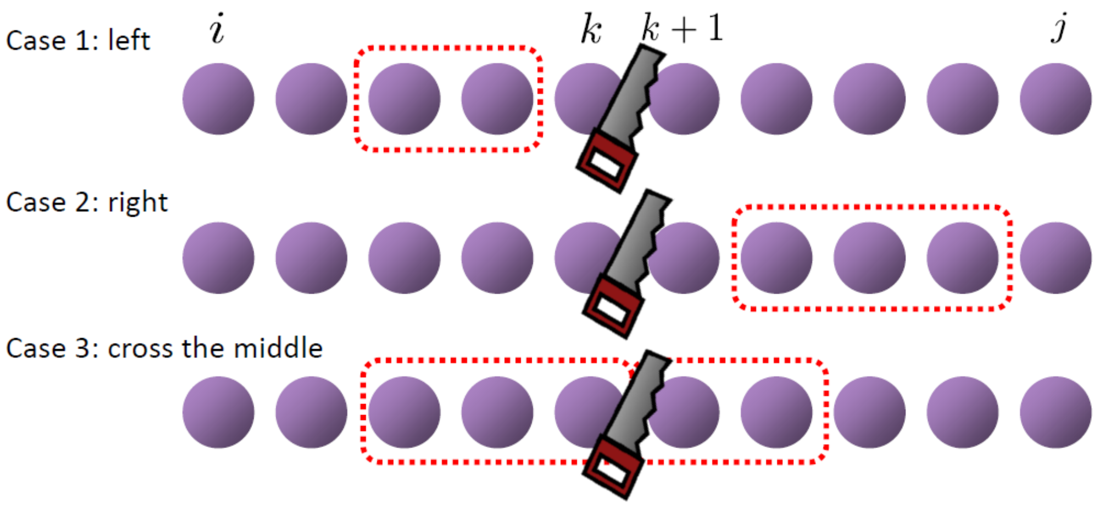
```
The array contains elements :(You can set it in the source code by yourself)

[ -24 0 66 -12 66 1 ]

Maximum sum in any contiguous suarray is 121

press any key to close 6356703
Process returned 0 (0x0)   execution time : 0.055 s
```

(2) array[] = { 156, -11, -39, 85, 22, -10 };

```
The array contains elements :(You can set it in the source code by yourself)

[ 156 -11 -39 85 22 -10 ]

Maximum sum in any contiguous suarray is 213

press any key to close 6356703
Process returned 0 (0x0)   execution time : 0.058 s
```

Algorithm

◎ The maximum subarray for any input must be in one of following cases:



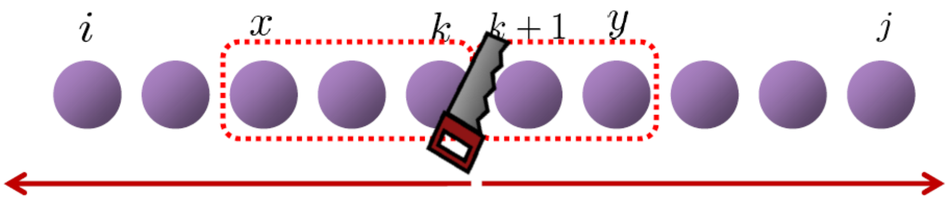
Case1: MaxSub(A, i, j) = MaxSub(A, i, k)

Case2: MaxSub(A, i, j) = MaxSub(A, k+1, j)

Case3: MaxSub(A, i, j) can not be expressed using MaxSub(),Instead it uses MaxCrossSub ,which is $\theta(n)$

◎ Case 3: Cross the Middle

-- Goal: find the maximum subarray that crosses the middle



(1) Start from the middle to find the left maximum subarray

(2) Start from the middle to find the right maximum subarray

The solution of Case 3 is the combination of (1) and (2)

-- Observation

- The sum of A[x…k] must be the maximum among A[i…k] (left: i<=k)
- The sum of A[k+1…y] must be the maximum among A[k+1…j] (right: j>k)
- Solvable in linear time → $\theta(n)$

◎ Divide-and-Conquer Algorithm

```
MaxCrossSubarray(A, i, k, j)
  left_sum = -∞
  sum=0
  for p = k downto i
    sum = sum + A[p]
    if sum > left_sum
      left_sum = sum
      max_left = p

  right_sum = -∞
  sum=0
  for q = k+1 to j
    sum = sum + A[q]
    if sum > right_sum
      right_sum = sum
      max_right = q
  return (max_left, max_right, left_sum + right_sum)
```

$O(k-i+1)$

$O(j-k)$

$= O(j-i+1)$

```
MaxSubarray(A, i, j)
  if i == j // base case
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)
    (l_low, l_high, l_sum) = MaxSubarray(A, i, k)
    (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)
    (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)

  if l_sum >= r_sum and l_sum >= c_sum // case 1
    return (l_low, l_high, l_sum)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2
    return (r_low, r_high, r_sum)
  else // case 3
    return (c_low, c_high, c_sum)
```

Divide

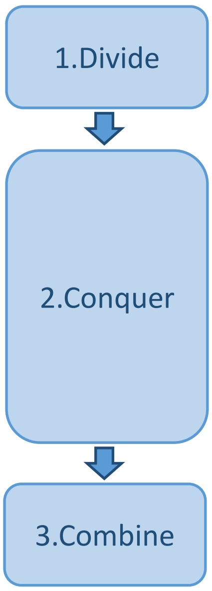
Conquer

Combine

```
MaxSubarray(A, i, j)                                                       O(1)
  if i == j // base case
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)
    (l_low, l_high, l_sum) = MaxSubarray(A, i, k)                          T(k − i + 1)
    (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)                        T(j − k)
    (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)
                                                                           O(j − i + 1)
  if l_sum >= r_sum and l_sum >= c_sum // case 1                           O(1)
    return (l_low, l_high, l_sum)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2                      O(1)
    return (r_low, r_high, r_sum)
  else // case 3                                                           O(1)
    return (c_low, c_high, c_sum)
```

(This is using Divide and Conquer approach,whereas it can reach to $O(n)$ by using Dynamic Programming approach)

Complexity



◎ Divide a list of size ***n*** into 2 subarrays of size ***n / 2*** .... $\theta(1)$

◎ Recursive case (n > 1) .... $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$

-- find MaxSub for each subarrays .... $\theta(1)$

◎ Base case (n = 1)

-- Return itself

◎ Find MaxCrossSub for the original list .... $\theta(n)$

◎ Pick the subarray with the maximum sum among 3 subarray .... $\theta(1)$

-- $T(n)$ = time for running MaxSubarray(A, i, j) with $j - i + 1 = n$

$$T(n) = \begin{cases} O(1), \ if\ n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \theta(n), \ if\ n \geq 2 \end{cases} \quad => \quad O(nlogn)$$

2. Write a program to calculate the Binomial Coefficient.

Test Result

(1) $C_2^5 = \frac{5\times4}{2\times1} = 10$

(2) $C_4^{15} = \frac{15\times14\times13\times12}{4\times3\times2\times1} = 1365$

```
Enter Binomial Coefficient (n>=k)

n= 5
k= 2

Binomial Coefficient C(5, 2)=10

press any key to close
```

```
Enter Binomial Coefficient (n>=k)

n= 15
k= 4

Binomial Coefficient C(15, 4)=1365

press any key to close
```

Algorithm

(1) Optimal Substructure

The value of $C(n,k)$ can be recursively calculated using following standard formula for Binomial Coefficients.

```
C(n, k) = C(n-1, k-1) + C(n-1, k)
C(n, 0) = C(n, n) = 1
```

(2) Overlapping Subproblems

It should be noted that the above function computes the same subproblems again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient problem has both properties of a dynamic programming problem. Like other typical dynamic programming problems, re-computations of same subproblems can be avoided by constructing a temporary array in buttom up manner.