1. Description and Analysis of Divide and Conquer

LS (left, right) Find the maximum sum of contiguous elements of the array A between indices *left* and *right*.

```
if left = right then
    return (left, left, A[left])
end
middle ← floor of (left + right)/2
(start1, end1, sum1) ← LS(b, middle)
(start2, end2, sum2) ← LS(middle+1, e)
leftSum ← Double.MIN_VALUE
startPoint ← middle
sum ← 0
for i=middle to left do
    sum ← sum + A[i]
    if sum > leftSum then
        leftSum ← sum
        startPoint ← i
    end
end
rightSum ← Double.MIN_VALUE
endPoint ← middle+1
sum ← 0
for i = middle+1 to right do
    sum ← sum + A[i]
    if sum > rightSum then
        rightSum ← sum
        endPoint ← i
    end
end
sum3 ← rightSum + leftSum
if sum1 = max {sum1, sum2, sum3} then
    return (start1, end1, sum1)
end
if sum2 = max {sum1, sum2, sum3} then
    return (start2, end2, sum2)
end
else return (startPoint, endPoint, sum3)
```

Time complexity:
A call to this algorithm with an array of size n leads to two recursive calls on arrays of size $n/2$. Dividing the array has a constant cost. Computing the overall results from the sub results requires scanning the whole array and hence has a cost of $\Theta(n)$. The complexity is $C(n) = 2C(n/2) + \Theta(n) = \Theta(n \log n)$

Space complexity:

No extra space required apart from making recursive stack calls, so space complexity is O (1).

2. Description and Analysis of Dynamic Programming

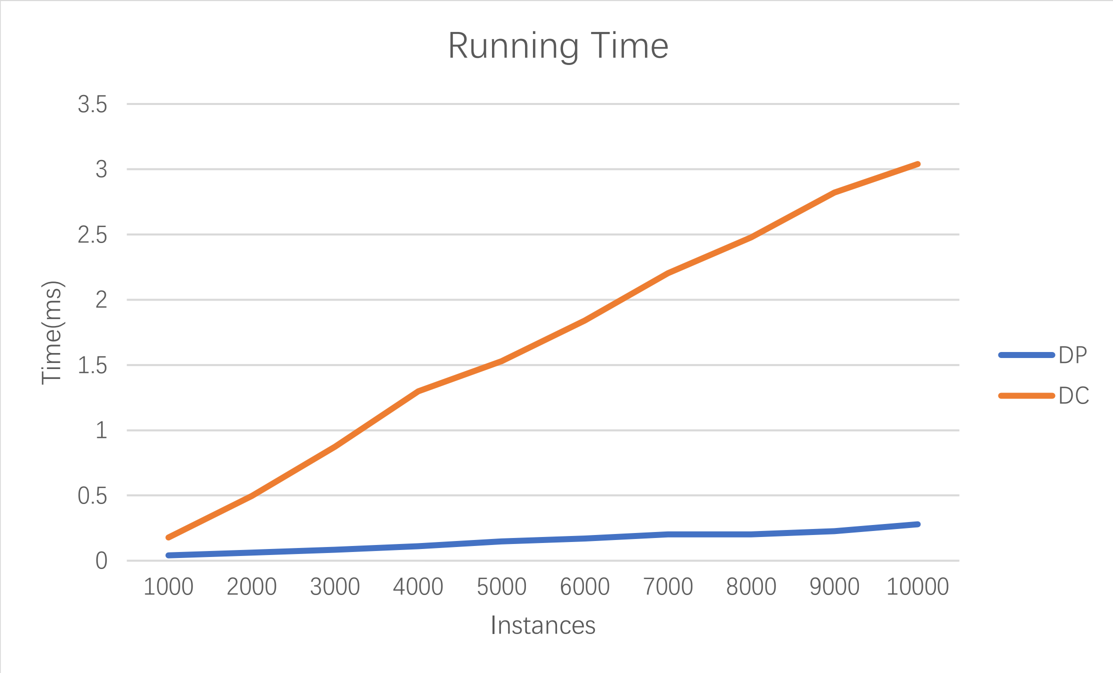
LS (left, right) Find the maximum sum of contiguous elements of the array A between indices *left* and *right*.

```
if left = right then
|  return (left, left, A[left])
end
listSum[0] ← A[left]
maxSum ← A[left]
for i=left+1 to right do
|   if listSum[i-1] <= 0 then
|   |  listSum[i] ← A[i]
|   end
|   if listSum[i-1] > 0 then
|   |  listSum[i] ← listSum[i-1] + A[i]
|   end
|   if listSum[i] > maxSum then
|   |  maxSum ← listSum[i]
|   |  end ← i
|   end
end
start ← end
for start = left+1 to end do
|   if listSum[start-1] <= 0 then
|   |  start = start+1
|   end
end
return (maxSum, start, end)
```

Time complexity:
There are three independent iterations in a call to this algorithm with an array of size *n*, with each iteration having at most *n* steps. So, the time complexity is O(n).

Space complexity:
Build an array to store maxSum for every i. Thus, space complexity is O(n).

3. Graph



4. Observation about the empirical results

The empirical results correspond to the analysis.
The time complexity of divide and conquer algorithm is O (n log n) which increases as the value of n increases. Because the time complexity of dynamic programming, O(n), is linear, the running time of DC increases much more than DP when n increases.

5. Comparison of two algorithms

In term of time complexity, dynamic programming takes O(n) while divide and conquer takes O(n log n). Dynamic programming algorithm is much faster than divide and conquer. As we can see from the plots that DP algorithm is a lot faster in comparison to DC and the difference in time keeps on increasing as we keep on increasing the problem size.
In term of space complexity, dynamic programming takes O(n) while divide and conquer takes O(1). Divide and conquer uses less space than dynamic programming.