Inversion Counting

* inversion number = # inversion pairs.

Theorem: # swaps in Insertion Sort. = Inversion Number.

proof. Assum theorem is correct for an array of size n-1

i.e. #swaps (Insertion Sort (A[1, n-1])) = Inversion number (A[1, n-1])

let x = A[n], Ramaing Work of Insertion Sorting:

1. Compare x to items in A[1, n-1]
2. Swap.

⇒ # Swaps in remaining Swaps

= # of items $j < n$ such that $A[j] > A[n]$

= # inversions related to x (A[n])

∴ in full Step. (n items), #swaps (Insertion Sort (A[1, n])) = Inversion number (A[1, n])

Algorithm: { 1. Check all pairs. ⇒ $\theta(n^2)$; 2. Run insertion Sort ⇒ $\theta(n^2)$

Divide-and-Conquer

```
Count ( A, p, q, r ):
L ← A[p, q] , R ← A[q+1, r]
L, R already sorted  ( Assumption )
i ← 1, j ← 1
while (i ≤ q-p+1) and ( j ≤ r-q )
    if L[i] ≤ R[j]: # no need to invert.
        i ← i+1
    else: # need to invert L[i...q] with R[j]
        I[j] = q-p-i+2 # elements in L[i...q]
        c ← c+I[j]
        j ← j+1
return c
```

As We're recursively calling Sort-and-Count ( ) itself.

L, R is indeed Sorted!

full inversion Counting algorithm:

```
Sort-and-Count ( A, p, r ):
    if p == r then return 0:
    q = (p+r)/2
    c1 ← Sort-and-Count ( A, p, q )
    c2 ← Sort-and-Count ( A, q+1, r )
    c3 ← Merge-and-Count (A, p, q, r)
    return c1 + c2 + c3
```

```
Merge & Count( A, p, r, mid )
L ← A[p...mid], R ← A[mid+1...r]
append ∞ at the end of L, R
i ← 1  j ← 1  c ← 0
for k ← p to r:
    if L[i] ≤ R[j] then:
        A[k] ← L[i], i ← i+1.
    else:
        A[k] ← R[j], j ← j+1
        c ← c + mid-p-i+2
return c.
```

Maximum Contiguous Subarray

Definition:

An array A[1...n]

Find the maximum V(i,j), Where $V(i,j) = \sum_{k=1}^{j} A[k]$

Brute force: traverse all subarray → $\theta(n^3)$

Data-reuse → $\theta(n^2)$

Divide-and-Conquer

Idea: 1. Divide into 2 halves

2. All subarrays can be classified into 3 cases:
   - ① entirely in L half
   - ② entirely in R half
   - ③ crosses L, R

3. find maximum in ① ② ③

* Maximums of ①, ② Can be found recursively

☆ Only need to consider Case ③.

Full Algorithm: $T(n) = \theta(n \log n)$

```
MaxSubarray (A, p, r):
if p == r then return A[p]
q = (p+r)/2.
M1 ← MaxSubarray (A, p, q)
M2 ← MaxSubarray (A, q+1, r)
Lm ← -∞, Rm ← -∞
V ← 0
for i ← q downto p:
    V ← V + A[i]
    if V > Lm then Lm ← V
V ← 0
for i ← q+1 to r:
    V ← V + A[i]
    if V > Rm then Rm ← V
return max{M1, M2, Lm+Rm}.
```

☆ for Case ③:

Maximum Subarray crossing Mid

= Max (A[i, q]) + Max (A[q+1, j])

⇒ We're going to find i, j

Then We can find maximum subarray in Case ③

Kadane's algorithm (Optimized Version: $T(n) = O(n)$)

```
Kadane-algorithm ( A, p, r ):
Vmax ← -∞ ; V ← 0 ; start ← 1 ; end ← 1 ; temp ← 1
for i ← p to r .do:
    V ← V + A[i]
    if A[i] > V then:
        V ← A[i]
        temp ← i # recording possible start position
    if V > Vmax then:
        Vmax ← V
        start ← temp , end ← i
return Vmax
```

Greedy

A[i] > V (V = Vpre + A[i]) ⇒ Vprevious < 0

⇒ We can restart from current position

☆ Observation:

if V[i-1] is negative

then any subsequence max subarray

Will not Contain this negative pare

Hence. We can restart from A[i]