Week 8

Friday, August 1, 2025 1:36 PM

Divide and Conquer: Break your problem into subtasks. Efficiently combine the solutions of the subtasks to find complete solution.

Ex: Merge & Quick sort

Counting Inversions: Inversions are cases when, index (i) < index (j). for i & j being elements of a given list but, value (i) > value (j).

Possible approaches to solve this →

1st Approach is to apply pointer at an index, and compare with all elements in the remaining part of list. Increment the count when condition fits.

2nd Approach is to make use of merge function of merge sort.

→ Number of inversions lie in the range of 0 to $\frac{n(n-1)}{2}$.

→ Recurrance relation: $2T(n/2) + n$

→ Complexity comes to be : $O(n \log n)$

Implementation →

→ merge 2 sorted lists and count the no. of inversions.

```
def mergeAndCount(A,B):
    (m,n) = (len(A),len(B))
    (C,i,j,k,count) = ([],0,0,0,0)
    while k < m+n:
        if i == m:
            C.append(B[j])
            j += 1
            k += 1
        elif j == n:
            C.append(A[i])
            i += 1
            k += 1
        elif A[i] < B[j]:
            C.append(A[i])
            i += 1
            k += 1
        else:
            C.append(B[j])
            j += 1
            k += 1
```

Length of left sub-list ↓ increment count by this

Inversion detected here. Since both lists are sorted then if there's an inversion at index 'i' all

```
        else:
            C.append(B[j])
            j += 1
            k += 1
            count += m-i
    return(C,count)
def inversionCount(A):
    n = len(A)
    if n <= 1:
        return(A,0)
    (L,countL) = inversionCount(A[:n//2])
    (R,countR) = inversionCount(A[n//2:])
    (B,countB) = mergeAndCount(L,R)
    return(B,countL + countR + countB)
```

increment count by m-i

index of current item

Inversion detected here. Since both lists are sorted then if there's an inversion at index 'i', all ahead of it will also have inversions

Count inversions in left half

Count inversions in right half

Count inversions b/w left & right half

Return sorted Array and inversion count. → Left half inversions → Right half inversions → in b/w inversions.

Closest Pair of Points → Several points on screen, having (x, y) coordinates. No 2 points have same x-coordinates. Aim is to find the closest pair.

Approaches → 1st Brute approach, by comparing every pair. Takes time $O(n^2)$.

2nd. Arrange in sorted order of x-coordinate. create a division boundary at middle co-ordinate. Keep doing this recursively. Find least dist. each side. take into consideration the min of them. Takes time $O(n \log n)$

→ Recurrance relation: $T(n) = 2T(n/2) + O(n)$

→ Best complexity: $O(n \log n)$

Implementation →

```
import math
# Returns eucledian disatnce between points p and q
def distance(p, q):
  return math.sqrt(math.pow(p[0] - q[0],2) + math.pow(p[1] - q[1],2))
def minDistanceRec(Px, Py):
  s = len(Px)
  # Given number of points cannot be less than 2.
  # If only 2 or 3 points are left return the minimum distance accordingly.
  if (s == 2):
    return distance(Px[0],Px[1])
  elif (s == 3):
    return min(distance(Px[0],Px[1]), distance(Px[1],Px[2]), distance(Px[2],Px[0]))
# For more than 3 points divide the poitns by point around median of x coordinates
  m = s//2
  Qx = Px[:m]
  Rx = Px[m:]
  xR = Rx[0][0]    # minimum x value in Rx

  # Construct Qy and Ry in O(n) rather from Py
  Qy=[]
```

For calculating euclidean dist. b/w 2 points.

Px: list of x coordinates

Py: list of y coordinates

Brute force approach if no of points less or 3

Recurrsively compute closest dist

```
  xR = Rx[0][0]   # minimum x value in Rx

  # Construct Qy and Ry in O(n) rather from Py
  Qy=[]
  Ry=[]
  for p in Py:
    if(p[0] < xR):
      Qy.append(p)
    else:
      Ry.append(p)
# Extract Sy using delta
  delta = min(minDistanceRec(Qx, Qy), minDistanceRec(Rx, Ry))
  Sy = []
  for p in Py:
    if abs(p[0]-xR) <= delta:
      Sy.append(p)

  #print(xR,delta,Sy)
  sizeS = len(Sy)
  if sizeS > 1:
      minS = distance(Sy[0], Sy[1])
      for i in range(1, sizeS-1):
          for j in range(i, min(i+15, sizeS-1)):
              minS = min(minS, distance(Sy[i], Sy[j+1]))
      return min(delta, minS)
  else:
      return delta
def minDistance(Points):
  Px = sorted(Points)
  Py = Points
  Py.sort(key=lambda x: x[-1])
  #print(Px,Py)
  return round(minDistanceRec(Px, Py), 2)
```

Recursively compute closest dist between pair of points.

Both in the left and right half planes created.

Creating the split strip.

Conservatively compare dist with next 15 points.

Recursive function to return min distance, rounded to 2 decimal points.

Integer multiplication

- Traditional method: $O(n^2)$
- Naïve divide and conquer strategy: $T(n) = 4T(n/2) + n = O(n^2)$
- Karatsuba's algorithm: $T(n) = 3Tn/2 + n \approx On\ log\ 3)$

Implementation

```
# here 10 represent base of input numbers x and y
def Fast_Multiply(x,y,n):
    if n == 1:
        return x * y
    else:
        m = n/2
        xh = x//10**m
        xl = x % (10**m)
        yh = y//10**m
        yl = y % (10**m)
        a = xh + xl
        b = yh + yl
        p = Fast_Multiply(xh, yh, m)
        q = Fast_Multiply(xl, yl, m)
        r = Fast_Multiply(a, b, m)
        return p*(10**n) + (r - q - p) * (10**(n/2)) + q
print(Fast_Multiply(3456,8902,4))
```

Half the number (should be $m = n//2$)

extract high part & low part for both the numbers created or obtained after the split.

Intermidiate sum using karatsuba's formula

Recursive calls.

karatsuba's formula to get the final multiplication result.

Output

```
30765312.0
```

Quick Select → Based on Quick sort's partitioning technique.

→ Aims to find the k^{th} largest element in the sequence of K elements.

- Sort in descending order and look at position k – $O(nlogn)$ → Brute force approach
- For any fixed k, find maximum for k times – $O(kn)$
- $k = n/2$ (median) – $O(n^2)$
- Median of medians –$O(n)$ → Quick select
- Selection becomes $O(n)$ in Fast select algorithm → Improved technique to find any element.
- Quicksort becomes $O(nlogn)$ using MoM → Improved Quick sort.

Median of Medians (Discussed Ahead)

Implementation →

```
def quickselect(L,l,r,k): # k-th smallest in L[l:r]
  if (k < 1) or (k > r-l):
    return(None)
(pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper + 1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      (lower,upper) = (lower+1,upper+1)
  (L[l],L[lower-1]) = (L[lower-1],L[l]) # Move pivot
  lower = lower - 1
# Recursive calls
  lowerlen = lower - l
  if k <= lowerlen:
    return(quickselect(L,l,lower,k))
  elif k == (lowerlen + 1):
    return(L[lower])
  else:
    return(quickselect(L,lower+1,r,k-(lowerlen+1)))
```

→ Boundary check.

Partitioning just like Quick sort.

Computes the lowerlength

If K is pivot, return pivot.

If k< pivot, recurse on left half.

If K>pivot, recurse on right half.

Master theorem (for finding time complexity from recurrance relations)

General Recurrance relation: $T(n) = aT(n/b) + O(n^k \log^p n)$

If ① $\log_b a > k \Rightarrow O(n^{\log_b a})$

② $\log_b a = k \Rightarrow O(n^k \log^{p+1} n)$

③ $\log_b a < k \Rightarrow O(n^k \log^p n)$

Qus. $T_1(n) = 3T_1(n/3) + O(n^2)$

Compare with $aT(n/b) + O(n^k \log^p n)$

$\Rightarrow a=3, b=3, K=2, P=0$

Ques.

$T_1(n) = 3T_1(n/3) + O(n)$

$T_2(n) = 9T_2(n/3) + O(n)$

Base case: $T_1(1) = T_2(1) = 1$

Compare with $aT(n/b) + O(n^k \log^p n)$

$\Rightarrow a = 3, b = 3, k = 2, p = 0$

$\Rightarrow \log_b a = \log_3 3 < k = 2$

$\Rightarrow$ Ans: $O(n^k \log^p k) \Rightarrow O(n^2)$ $\rightarrow$ for T_1

for $T_2 \rightarrow a = 9, b = 3, k = 1, p = 0$ $\Rightarrow \log_3 9 = 2 > k = 1$

$\Rightarrow$ Ans: $O(n^{\log_b a}) = O(n^2)$ $\rightarrow$ for T_2

Ques.

Q

A $T(n) = 2T(n/4) + O(n)$

B $T(n) = 3T(n/3) + O(n)$

C $T(n) = 9T(n/3) + O(n)$

D $T(n) = T(n/2) + O(1)$

FOR (A); $a = 2, b = 4, k = 1, p = 0$

$\log_4 2 = \frac{1}{2} < k \Rightarrow O(n)$

FOR (B); $a = 3, b = 3, k = 1, p = 0$

$\log_3 3 = 1 = k = 1 \Rightarrow O(n \log n)$

FOR (C); $a = 9, b = 3, k = 1, p = 0$

$\Rightarrow \log_3 9 = 2 > k = 1 \Rightarrow O(n^{\log_3 9}) = O(n^2)$

FOR (D); $a = 1, b = 2, k = 0, p = 0$

$\Rightarrow \log_2 1 = 0 = k = 0 \Rightarrow O(\log n)$

Median of Medians →

1. Make blocks of the given length.
2. Sort every block.
3. Find the middle of each block.
4. Repeat this until the size of array of medians is less than the block length.
5. Find the middle term of the medians array. (sorted array)

If the last block has lesser length than desired, don't worry, just follow with the steps.

Imp. Always sort the array of medians, before reporting answer.

Aim is to select a good pivot.

Implementation →

For length less than 5 {

```
def MoM(L): # Median of medians
  if len(L) <= 5:
    L.sort()
    return(L[len(L)//2])
  # Construct list of block medians
```

Implementation

For length less than 5

```
  if len(L) <= 5:
    L.sort()
    return(L[len(L)//2])
  # Construct list of block medians
  M = []
  for i in range(0,len(L),5):
    X = L[i:i+5]
    X.sort()
    M.append(X[len(X)//2])
  return(MoM(M))
```

create blocks of length 5, sort each block, get middle term, create median array, Repeat until median array < block length.

Return Median of Medians.