Divide and Conquer Examples

Table of Contents:

Master's Theorem

- Let $a \geq 1$ and $b > 1$ be constants.
  Let $f(n)$ be an asymptotically positive function.
  Let $T(n) \leq aT(\frac{n}{b}) + O(f(n))$ be a recurrence relation.

  Then:
  1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = O(n^{\log_b a})$
  2. If $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ for some constant $k \geq 0$, then $T(n) = O(n^{\log_b a} \cdot \log^{k+1} n)$
  3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and f satisfies the regularity condition then $T(n) = O(f(n))$

     The regularity condition states that:
     For some constant $c < 1$, and all sufficiently large n, $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$

- Examples:
  1. $T(n) = 9T(\frac{n}{3}) + n$

     Soln:

     $a = 9$
     $b = 3$
     $d = \log_b a$
     $= \log_3 9$
     $= 2$
     $f(n) = n$
     $n^d = n^2$

     Based on case 1, where $\varepsilon = 1$,
     $T(n) = O(n^2)$

2. $T(n) = T(\frac{2n}{3}) + 1$

Soln:

$a = 1$

$b = \frac{3}{2}$

$d = \log_b a$

$\quad = 0$

$f(n) = 1$

$n^d = n^0$

$\quad = 1$

Based on case 2, where $k=0$,
$T(n) = O(\log n)$

3. $T(n) = 3T(\frac{n}{4}) + n\lg n$

Soln:

$a = 3$

$b = 4$

$d = \log_b a$

$\quad = \log_4 3 \approx 0.79$

$f(n) = n\lg n$

$n^d = n^{\log_4 3}$

$\quad \approx n^{0.79}$

We see that $\varepsilon \approx 0.2$ as $n^{0.79+0.2} \approx n$.
Now, we have to prove that f satisfies the regularity condition.

a. $f(\frac{n}{b}) \leq c \cdot f(n)$, for some constant $c<1$

$3(\frac{n}{4} \cdot \log(\frac{n}{4})) \leq c \cdot n\lg n$

$3(\frac{n}{4}(\log n - \log 4)) \leq c \cdot n\lg n$

$\frac{3n}{4}\log n - \frac{3n}{4}\log 4 \leq c \cdot n\lg n$

We see that $c = 3/4$ satisfies it.

4. $T(n) = 2T(\frac{n}{2}) + n\lg n$

Soln:

$a=2$

$b=2$

$d = \log_b a = 1$

$f(n) = n\lg n$

$n^d = n$

We can't use Master's Theorem here because lg n is not polynomially bigger than n, so we can't find an ε to satisfy case 3.

5. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

Soln:

$a=2$

$b=4$

$d = \log_b a = \frac{1}{2}$

$f(n) = \sqrt{n} = n^{1/2}$

$n^d = n^{1/2}$

Use case 2, $k=0$.

$T(n) = O(\sqrt{n} \cdot \lg n)$

6. $T(n) = 2T(\frac{n}{4}) + n^2$

Soln:

$a = 2$

$b = 4$

$d = \log_b a = \frac{1}{2}$

$f(n) = n^2$

$n^d = n^{1/2}$

Use case $3 \rightarrow \varepsilon = \frac{3}{2}$

Regularity Condition Proof

$2(\frac{n}{4})^2 \leq c \cdot n^2$

$\frac{2n^2}{16} \leq c \cdot n^2$

$c = \frac{2}{16}$

$\therefore T(n) = O(n^2)$

Max Subarray Problem

- Problem: Given an array of ints, we want to find the max sum of any valid subarray.

- Solution 1 (Naive Way):
  The "naive" approach is to loop the array for each element in the array and find the sum each time and update the max sum if the sum is bigger than it.

  Time Complexity: $O(n^2)$

  Code:

```
def maxSubArray(a):
    max_sum = -inf

    for i in range(len(a)):
        sum = 0
        for j in range(i, len(a)):
            sum += a[j]
            max_sum = max(max-sum, sum)


    return max-sum
```

- Solution 2 (Divide and Conquer)

The divide and conquer way is to break the array into halves, find the max subarray value of each halve and then see if a bigger subarray sum that spans the 2 halves exist.

Time Complexity: O(nlgn)

Code:

```
def MSA(a):
    if len(a) < 2:
        return a[0]


    mid = len(a)//2

    return max(MSA(a[:mid+1]),
               MSA(a[mid+1:]),
               merge(a, mid)
              )

def merge(array, mid):
    curSum = 0
    maxLSum = -inf

    for i in range(mid, -1, -1):
        curSum += array[i]
        maxLSum = max(curSum, maxLSum)
```

```
curSum = 0
maxRSum = -inf
for i in range(mid+1, len(array)):
    curSum += array[i]
    maxRSum = max(curSum, maxRSum)

return max(maxLSum, maxRSum,
           maxLSum + maxRSum
          )
```

- Solution 3 (Kadane's Algorithm):

Time Complexity: O(n)

Code:

```
def MSA(a):
    curSum = 0
    maxSum = -inf

    for num in a:
        curSum += num
        maxSum = max(maxSum, curSum)

    return maxSum
```

Strassen's Algorithm:

- Used to multiply matrices (Suppose that both matrices are nxn where n is a power of 2.)

- Solution 1 (Naive Approach):
  Here, we're using the traditional approach for multiplying matrices.

  Time Complexity: $O(n^3)$

  Code:

```
def MM(a1, a2):
    new_arr = []   ← Assume we initalized new_arr properly (O(n^2)).
    N = len(a1)

    for i in range(N):
        for j in range(N):
            for k in range(N):    Gets the elements of the rows
                new_arr[i][j] += a1[i][k] x a2[k][j]
                                             Gets the elements of the cols

    return new_arr
```

- Solution 2 (Divide and Conquer)

Here, we aim to use divide and conquer to solve the problem.

We will recursively divide the array into $\frac{n}{2} \times \frac{n}{2}$ arrays until the dimensions are at most 2x2.

Then, we'll merge the different pieces together.

Time Complexity: $O(n^3)$

Code:

```
def MM (A,B):
    N= len(A)

    if N==1:
        return A[0][0] x B[0][0]

    Compute A11, A12, A21, A22, B11, B12, B21, B22

    C1 = MM (A11, B11)
    C2 = MM (A11, B21)
    C3 = MM (A12, B12)
    C4 = MM (A12, B22)
    C5 = MM (A21, B11)
    C6 = MM (A21, B21)
    C7 = MM (A22, B12)
    C8 = MM (A22, B22)

    return [[[C1+C2], [C3+C4]], [[C5+C6], [C7+C8]]]
```

Consider 2 2x2 matrices A and B

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

We know that:

$$\begin{bmatrix} \underbrace{A_{11} \cdot B_{11}}_{C_1} + \underbrace{A_{12} \cdot B_{21}}_{C_2} & \underbrace{A_{11} \cdot B_{12}}_{C_3} + \underbrace{A_{12} \cdot B_{22}}_{C_4} \\ \underbrace{A_{21} \cdot B_{11}}_{C_5} + \underbrace{A_{22} \cdot B_{21}}_{C_6} & \underbrace{A_{21} \cdot B_{12}}_{C_7} + \underbrace{A_{22} \cdot B_{22}}_{C_8} \end{bmatrix}$$

For $n > 2$, we just recurse.

Time Complexity:

Notice we make 8 recursive calls.

Furthermore, we're multiplying 2 $\frac{n}{2} \times \frac{n}{2}$ matrices with each recursive call, so we get $T(\frac{n}{2})$. ← n is the length of the array.

Lastly, adding the matrices take $\Theta(n^2)$ time.

Overall, we get $T(n) = 8\,T(\frac{n}{2}) + \Theta(n^2)$.

Use Master's Thm:

$a = 8$

$b = 2$

$d = \log_b a = 3$

$f(n) = \Theta(n^2)$

$n^d = n^3$

Use case 1 → $T(n) = O(n^3)$

— Solution 3 (Strassen's Algo):

Strassen removed 1 recursive call by using more additions/subtractions.

$P_1 = A_{11}(B_{12} - B_{22})$
$P_2 = (A_{11} + A_{12})B_{22}$
$P_3 = (A_{21} + A_{22})B_{11}$
$P_4 = A_{22}(B_{21} - B_{11})$
$P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$
$P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$
$P_7 = (A_{11} - A_{21})(B_{11} + B_{12})$

$C_1 = P_5 + P_4 - P_2 + P_6$
$C_2 = P_1 + P_2$
$C_3 = P_3 + P_4$
$C_4 = P_1 + P_5 - P_3 - P_7$

Time Complexity: $O(n^{\lg_2 7})$

Code:

```
def Strassen(A,B):
    N = len(A)
    if N == 1:
        return A[0][0] x B[0][0]
    Compute A11, A12, A21, A22, B11, B12, B21, B22
    P1 = Strassen(A11, B12 - B22)
    P2 = Strassen(A11 + A12, B22)
            ⋮
    P7 = Strassen(A11 - A21, B11 + B12)
    C1 = P5 + P4 - P2 + P6
    C2 = P1 + P2
    C3 = P3 + P4
    C4 = P1 + P5 - P3 - P7
    return [[C1, C2], [C3, C4]]
```

Time Complexity:

$T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$

$a = 7$

$b = 2$

$d = \log_b a = \log_2 7$

$f(n) = n^2$

$n^d = n^{\log_2 7}$

$\approx n^{2.81}$

Use case 1 of Master's Theorem.

$T(n) = O(n^{\lg_2 7})$

$\approx O(n^{2.81})$

Karatsuba's Algo

- Used to multiply 2 n-digit ints, both in base r.

- Solution 1 (Naive Approach):

  This is the algorithm taught in school.

  Time Complexity: $O(n^2)$

- Solution 2 (Divide and Conquer):

  We divide each number into 2 halves

  $x = x_H r^{n/2} + x_L$

  $y = y_H r^{n/2} + y_L$

  Then
  $$\begin{aligned} xy &= (x_H r^{n/2} + x_L)(y_H r^{n/2} + y_L) \\ &= x_H r^{n/2} y_H r^{n/2} + x_H r^{n/2} y_L + \\ &\quad y_H r^{n/2} x_L + x_L y_L \\ &= (x_H y_H) r^n + x_H r^{n/2} y_L + \\ &\quad y_H r^{n/2} x_L + x_L y_L \\ &= (x_H y_H) r^n + (x_H y_L + y_H x_L) r^{n/2} + x_L y_L \end{aligned}$$

  Since there are 4 multiplications, there are 4 recursive calls.

  Time Complexity: $O(n^2)$

  $T(n) = 4T(\frac{n}{2}) + O(n)$

  $a = 4$

  $b = 2$

  $d = \log_b a = \log_2 4 = 2$

  $f(n) = O(n)$

  Use Master's Thm $\rightarrow$ Case 1

  $\therefore \; T(n) = O(n^2)$

Code:

```
def Multiply (a, b):
    if (len(a) == 1):
        return a x b

    n = len(a) // 2
    aL = a[:n] · 10^(n/2)   <- Assume a and b are base 10
    aR = a[n:]
    bL = b[:n] · 10^(n/2)
    bR = b[n:]

    return Multiply (aL, bL) +
           Multiply (aL, bR) +
           Multiply (aR, bL) +
           Multiply (aR, bR)
```

We need to multiply a_L and b_L by $10^{n/2}$.
E.g. 1234
$\rightarrow \underbrace{12 \times 10^2}_{a_L} + \underbrace{34}_{b_L}$

- Solution 3 (Karatsuba's Algo):
  Uses 3 recursive calls instead of 4.
  $a = x_H y_H$
  $d = x_L y_L$
  $e = (x_H + x_L)(y_H + y_L) - a - d$
  Then, $xy = a \cdot r^n + e \cdot r^{n/2} + d$

  Time Complexity: $O(n^{\log_2 3}) \approx O(n^{1.584})$
  $T(n) = 3T(\frac{n}{2}) + O(n)$
  $a = 3$
  $b = 2$
  $d = \log_b a$
  $= \log_2 3$
  $n^d = n^{\log_2 3}$
  $f(n) = O(n)$
  By Master's Thm $\rightarrow$ Case 1 $\rightarrow$ $T(n) = O(n^{\lg_2 3})$

Code:

```
def Multiply (a, b):
    if (len(a) == 1):
        return axb

    n = len(a)//2
    X_H = a[:n]·10^(n/2)
    X_L = a[n:]
    Y_H = b[:n]·10^(n/2)
    Y_L = b[n:]
    a =     multiply(X_H, Y_H)
    d =     multiply (X_L, Y_L)

    e = multiply((X_H+X_L), (Y_H+Y_L))-a-d
    return a+d+e
```

Counting Inversions:

- Given an array a of length n, count the number of pairs (i,j) s.t. $i<j$ but $a[i] > a[j]$.

- Naive Soln:

  Loop through the array for each element in the array and check how many inversions there are.

  Time Complexity: $O(n^2)$

  Code:

```
def countingInversions(a):
    num_inversions = 0

    for i in range(len(a)):
        inversions = 0
        for j in range(i+1, len(a)):
            if (a[i] > a[j]):
                inversions += 1
        num_inversions += inversions

    return num_inversions
```

- Divide and Conquer Soln:

  Divide the array into half and count the number of inversions in each half as well as the number of inversions straddling btwn the halves. For that last bit, we'll sort each half to make it easier.

  Time Complexity: $O(n\lg n)$

Code:

```
def counting-inversion(a):
    n=len(a)
    if(n==1):
        return (a,0)

    mid = n//2
    left, left-inv = counting-inversion (a[:mid])
    right, right-inv = c-i(a[mid:])
    merge, merge-inv = merge-inv(left, right)

    return (merge, left-inv + right-inv + merge-inv)

def merge-inversion(a1, a2):
    a1 = a1.sort()  } We can sort bc we know all
    a2 = a2.sort()  } indexes of a1 < indexes of a2
    num-inv = 0

    for num in a2:
        idx = binary-search(a1, num) ← Find the index of
                                       the biggest element smaller
                                       than or equal to num.
        num-inv += len(a1)-idx -1
        if(len(a1) - idx -1 ==0):
            break

    return (a1+a2, num-inv)
```

This means that there are no elements in a_1 bigger than num. Since a_2 is sorted, we know future numbers in a_2 > all numbers in a_1.

Closest Pair in R^2:

- Given n points in the form (x_i, y_i), find the closest pair of points.

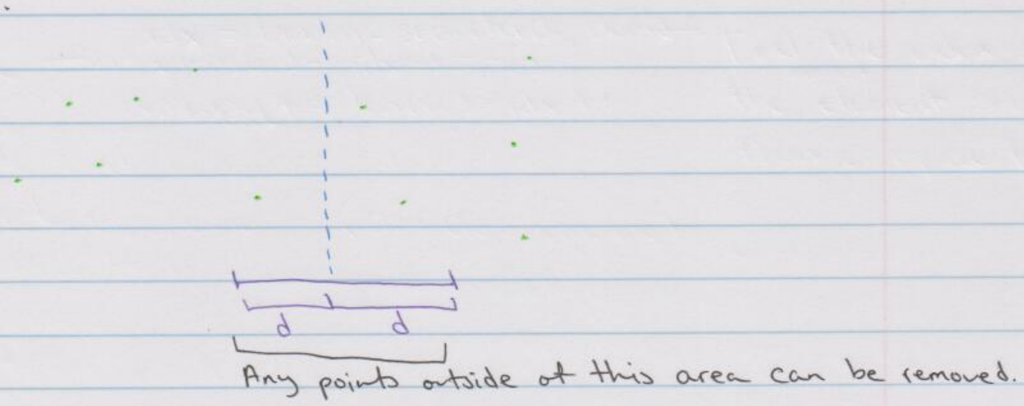
- Naive Soln:
  Compare each point with every point and calculate the distance and store the min distance.

  Time Complexity: $O(n^2)$

- Divide and Conquer Soln:
  Divide the set of points into halves and find the shortest distance on each side. Let d be the shortest distance from both sides.

  Then, when we see if there's points that straddle btwn the divided line whose distance may be smaller, we just have to check the points that are within d distance of the divided line.

  I.e.



Time Complexity: $O(n \lg^2 n)$

Quick Sort

- Chooses a pivot and then puts all elements less than or equal to the pivot on the left side and all other elements on the right side.
- Time Complexity: O(nlgn)
- Code:

```
def QS(a):
    if (len(a) == 1):
        return a

    pivot = a[-1]
    l = 0
    r = len(a) - 2
    while (l<r):
        if (a[l] > pivot and a[r] ≤ pivot):
            swap(a[l], a[r])
            l+=1
            r-=1
        elif (a[l] ≤ pivot and a[r] ≤ pivot):
            l+=1
        elif (a[l] > pivot and a[r] > pivot):
            r-=1
        else:
            l+=1
            r-=1

    if (l==r):
        if (a[l] ≤ pivot):
            l+=1

    return QS(a[:l]) + [pivot] + QS(a[l:])
```

Merge Sort

- Divides the array into halves, recursively.
- Combines the halves by comparing the elements.
- Time Complexity: O(nlgn)
- Code:

```
def MS(a):
    if (len(a) == 1):
        return a

    mid = len(a)//2
    left = MS(a[:mid])
    right = MS(a[mid:])
    return merge(left, right)

def merge(l, r):
    new_arr = []

    while(l and r):
        if (l[0] < r[0]):
            new_arr.append(l.pop(0))
        elif (l[0] > r[0]):
            new_arr.append(r.pop(0))
        else:
            new_arr.append(l.pop(0))
            new_arr.append(r.pop(0))

    if(l):
        new_arr += l

    if(r):
        new_arr += r

    return new_arr
```