Divide-and-Conquer

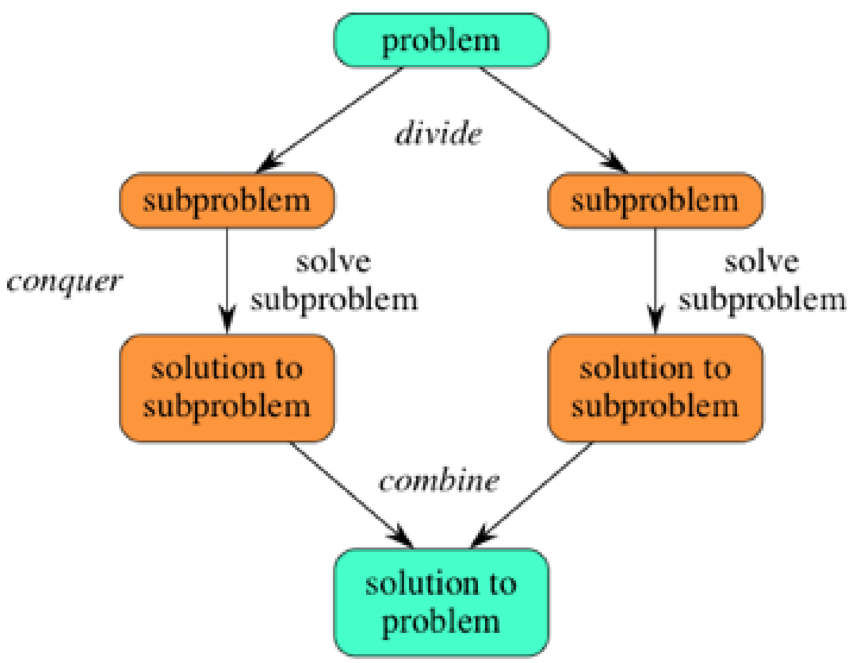
For divide-and-conquer, you solve a given problem (instance) recursively. If the problem is small enough - the base case - you just solve it directly without recursing. Otherwise - in the recursive case - you perform three characteristic steps:

- Divide - Divide the problem into one or more subproblems that are smaller instances of the same problem.
- Conquer - Conquer the subproblems by solving them recursively.
- Combine - Combine the subproblem solutions to form a solution to the original problem.

A divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which themselves may be broken down into even smaller subproblems, and so forth. The recursion bottoms out when it reaches a base case and the subproblem is small enough to solve directly without further recursing.

To recap, here's how divide-and-conquer works:
1. Figure out a simple case as the base case.
2. Figure out how to reduce your problem and get to the base case.

Divide-and-conquer isn't a simple algorithm that you can apply to a problem. Instead, it's a way to think about a problem.



Maximum Subarray Problem:

Input: An array A[1..n] of n numbers where numbers can be negative
Output: A non-empty subarray A[i..j] having the largest sum $S[i,j] = a_i + a_{i+1} + \ldots + a_j$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

maximum
subarray

- Brute Force Solution:

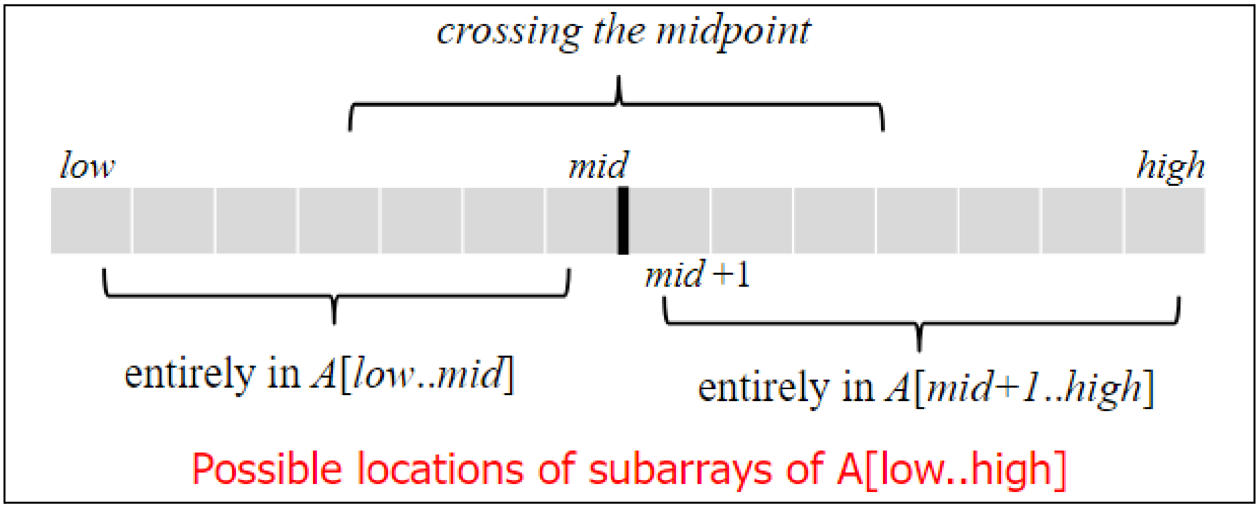
Try out all possible contiguous subarrays.

$$
\begin{array}{r}
A[1..1], A[1..2], A[1..3], \ldots, A[1..(n\text{-}1)], A[1..n] \\
A[2..2], A[2..3], \ldots, A[2..(n\text{-}1)], A[2..n] \\
\ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\
A[(n\text{-}1)..(n\text{-}1)], A[(n\text{-}1)..n] \\
A[n..n]
\end{array}
$$

```
max = -∞
for i = 1 to n do
begin
   sum = 0
   for j = i to n do
   begin
      sum = sum + A[j]
      if sum > max
      then max = sum
   end
end
```

Time Complexity: $O(n^2)$

- Divide-and-Conquer Solution:



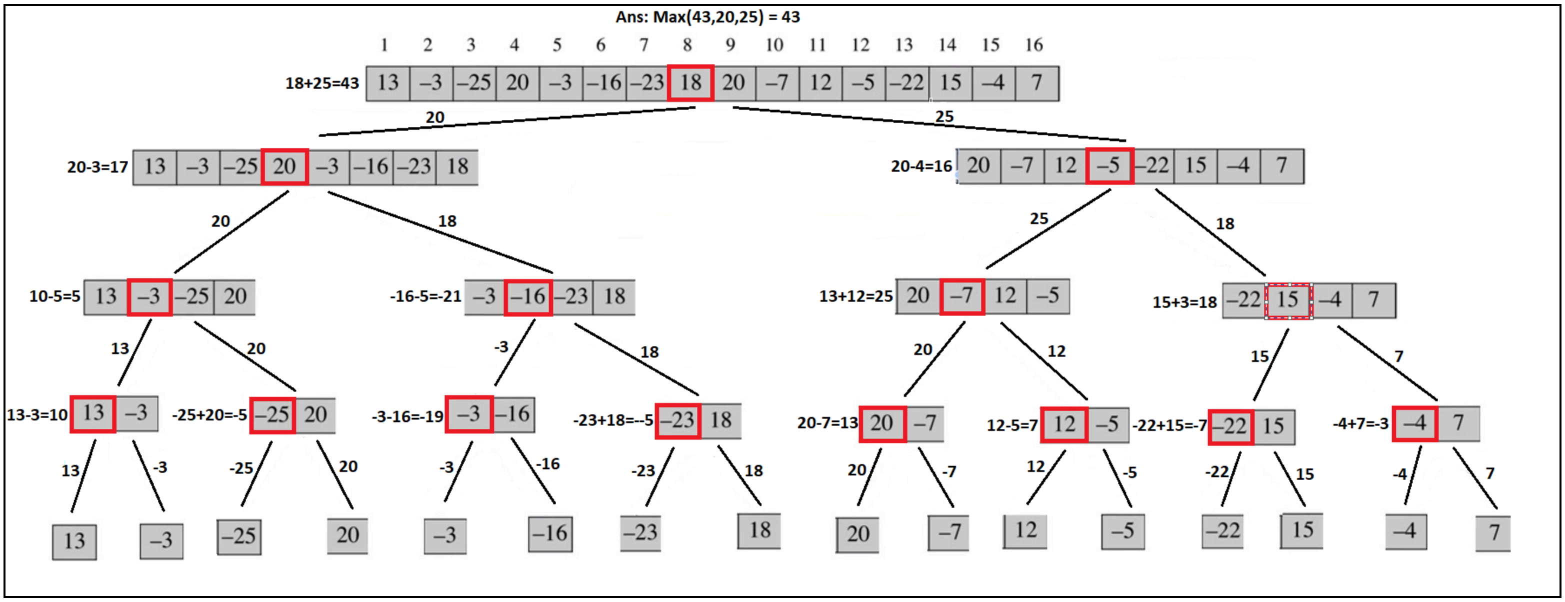
1) Divide the given array into two halves
2) Return the max of the following three:
    a) Recursively calculate max subarray in left half
    b) Recursively calculate max subarray in right half
    c) Recursively calculate max subarray sum such that the subarray crosses midpoint

        i) Find the maximum sum starting from mid point and ending at some point on left of mid

        ii) Find the maximum sum starting from mid+1 point and ending at some point on right of mid+1

        iii) Finally, combine the two and return

Simulation: (Zoom in to see clearly)



Pseudo Code:

```
def MAX-CROSS-SUBARRAY(A, low, mid, high)
        left-sum = -∞ # Find a maximum subarray of the form A[i..mid]
        sum = 0
        for i = mid down to low
                sum = sum + A[i]
                if sum > left_sum
                        left_sum = sum
                        max_left = i

        right_sum = -∞ #Find max subarray in A[mid + 1 .. j ]
        sum = 0
        for j = mid +1 to high
                sum = sum + A[j]
                if sum > right-sum
                        right_sum = sum
                        max_right = j

       return left_sum + right_sum
```

```
def MAXIMUM-SUBARRAY(A, low, high)
        if high == low
            return A[low]      // base case: only one element

        else mid = (low + high) // 2
             max_left_sum = MAXIMUM-SUBARRAY(A, low, mid)
             max_right_sum =  MAXIMUM-SUBARRAY(A, mid + 1, high)
             max_cross_sum = MAX-CROSS-SUBARRAY(A, low, mid,high)
             return max(max_left_sum, max_right_sum, max_cross_sum)
```

Follow-up Question:

Q) What are the Recurrence Equation and Time Complexity of this algorithm?

Ans:

FIND-MAX-CROSSING-SUBARRAY : $\Theta(n)$,
where $n = high - low + 1$

FIND-MAXIMUM-SUBARRAY

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Karatsuba's Multiplication Algorithm:

Input: Two n-digit numbers, x and y
Output: Product of x and y

- Brute Force Solution:

```
     1 2 3 4      -----> x
   x 8 7 6 5      -----> y
   -----------
     6 1 7 0      -----> n operations
   7 4 0 4 -      -----> n operations
 8 6 3 8 - -      -----> n operations
9 8 7 2 - - -     -----> n operations
---------------
1 0 8 1 6 0 1 0
```

Each digit multiplication is a single operation. If there are n digits in x, then there are n operations for each digit in y. If there are n digits in y, total operations would be n*n = n^2.

Time Complexity: $O(n^2)$

- Divide-and-Conquer Solution:

The key to understanding Karatsuba's multiplication algorithm is remembering that you can express *x* (an n-digit integer) in the following way:

$$x = a \times 10^{n/2} + b$$

For example you can express 2925 as:

$$\begin{aligned} 2925 &= 29 \times 100 + 25 \\ &= 29 \times 10^2 + 25 \end{aligned}$$

You can use this if you want to multiply *x* by another n-digit integer *y*:

$$y = c \times 10^{n/2} + d$$

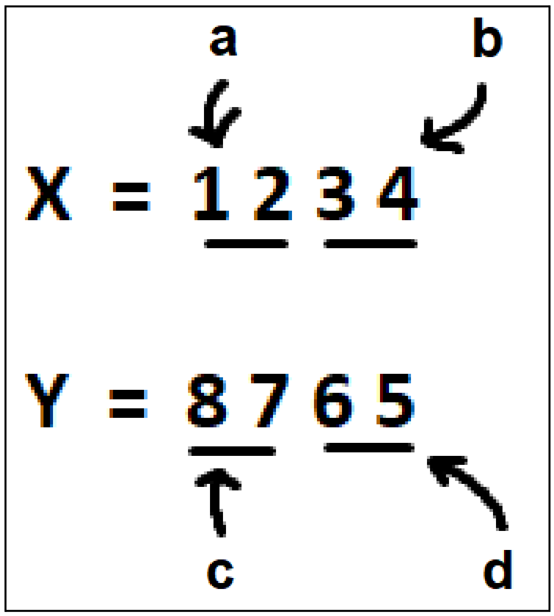
Then *x* multiplied by *y* can be written as:

$$\begin{aligned} xy &= (a \times 10^{n/2} + b)(c \times 10^{n/2} + d) \\ &= ac \times 10^n + (ad + bc) \times 10^{n/2} + bd \end{aligned}$$

This is where Karatsuba found a neat trick. He found a way to calculate *ac*, *bd* and *(ad + bc)* with just three multiplications (instead of four).

$$ad + bc = (a+b)(c+d) - ac - bd$$

ac and bd are already calculated. (a+b)(c+d) needs only 1 multiplication. Instead of multiplying two n/2 digit numbers 4 times, we can calculate the same result by multiplying two n/2 digit numbers 3 times. If we didn't make this improvement, then 4 multiplications would lead to $O(n^{\log 4}) = O(n^2)$, which is the same as normal multiplication.

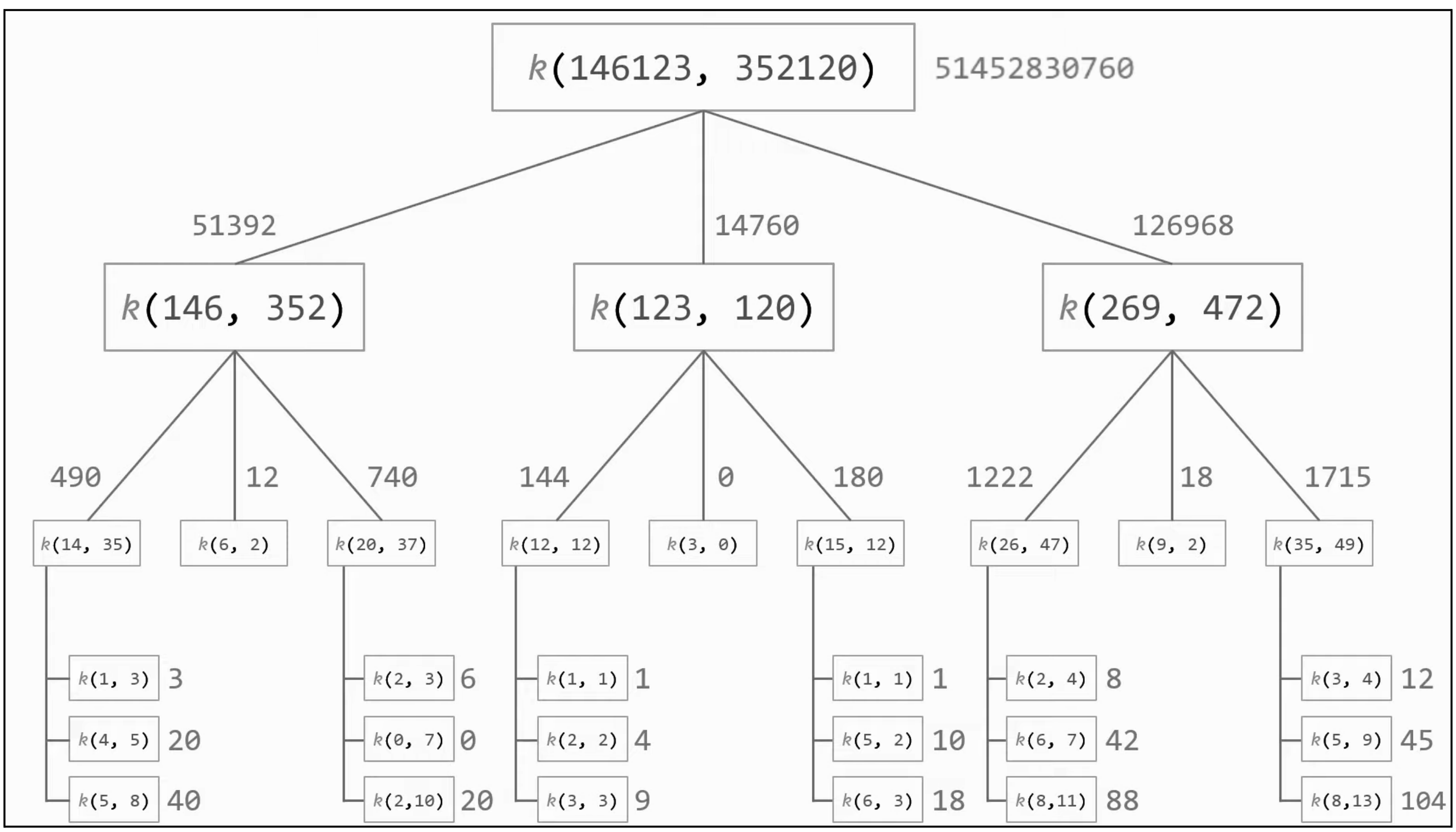


1) Divide the two numbers x and y into halves
2) Recursively Compute a * c (S_1)
3) Recursively Compute b * d (S_2)
4) Recursively Compute (a + b) * (c + d) (S_3)
5) Compute $S_3 - S_2 - S_1$ (S_4)
6) Return $S_1 * (10^n) + S_4 * (10^{n/2}) + S_2$

Pseudo Code:

```
def KARATSUBA(x, y):
        if x<10 or y<10:
                return x * y
        n = max(len(x), len(y))
        mid = n // 2
        a = x // 10**mid
        b = x % 10**mid
        c = y // 10**mid
        d = y % 10**mid
        S1 = KARATSUBA(a, c)
        S2 = KARATSUBA(b, d)
        S3 = KARATSUBA(a+b, c+d)
        S4 = S3 - S2 - S1
        return S1 * 10**n + S4 * (10**mid) + S2
```

Simulation: (Zoom in to see clearly)



Follow-up Question:

Q) What is the Recurrence Equation?

Ans: Depends on the number of subproblems and each subproblem size and work done at each step. In each step, there are 3 subproblems where size gets divided by 2 (n/2) and work done is n at each step due to adding or subtracting 2 n-bit numbers. $T(n) = 3T(n/2) + n$

Q) What is the Time Complexity?

Ans: $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Q) Can we use this on other number-based representations such as binary?

Ans: Yes, just replace base 10 with base 2 in the calculations.